University of Alberta

FREQUENT SETS BY LEAP-TRAVERSAL FOR SEQUENTIAL AND PARALLEL PARADIGMS

by

Mohammad Omar El-Hajj

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 2006

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-23024-4
Our file Notre référence
ISBN: 978-0-494-23024-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Efficient discovery of frequent patterns from large databases is an active research area in data mining with broad applications in industry and deep implications in many areas of data mining. Although many efficient frequent-pattern mining techniques have been developed in the last decade, most of them assume relatively small databases, leaving extremely large but realistic datasets out of reach. When computationally feasible, mining extremely large databases produces tremendously large numbers of frequent patterns. Mining for frequent itemsets can generate an overwhelming number of patterns, often exceeding the size of the original transactional database. In many cases, it is impractical to mine those datasets due to their sheer size; not only because of the extent of the existing patterns, but mainly the magnitude of the search space.

In this research we propose a new traversal approach that jumps in the search space among only promising nodes. Our leaping approach avoids nodes that would not participate in the answer set and drastically reduces the number of candidate patterns. We use this approach to efficiently pinpoint maximal patterns at the border of the frequent patterns in the lattice and collect enough information in the process to generate all subsequent patterns.

Using this approach we did mine sequentially sizes never been reported. We also generated different types of patterns and push constraints efficiently to filter the answer set to only patterns that are of interest to the decision makers.

To open the doors to the mining of extremely large databases, parallelizing the search for frequent patterns plays an important role. Not all good sequential algorithms can be effectively parallelized and parallelization alone is not enough. An algorithm has to be well suited for parallelization, and in the case of frequent pattern mining, clever methods for searching are certainly an advantage. The algorithm we propose for parallel mining of frequent maximal patterns, is based on our new technique for astutely jumping within the search space, and more importantly, is composed of autonomous task segments that can be performed separately and thus minimize communication between processors. Our parallel algorithm for mining frequent patterns generates all types of patterns and supports constraints pushing. Using this approach allows the mining, in a reasonable time, of databases in the order of billion transactions using relatively inexpensive clusters.

1) READ in the name of thy Sustainer, who has created
(2) created man out of a germ-cell!
(3) Read - for thy Sustainer is the Most Bountiful One
(4) who has taught [man] the use of the pen
(5) taught man what he did not know!
(6) Nay, verily, man becomes grossly overweening
(7) whenever he believes himself to be self-sufficient:
The Message of The Quran by Muhammad Asad
Sura 96 AL-ALAQ (THE GERM-CELL), Verses 1-7

To my parents

Wife
and
kids

Contents

1 Introduction 2
- 1.1 Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Thesis Statement . . . 4
- 1.4 Thesis Contributions . . . 4
- 1.5 Research Methodology . . . 5
- 1.6 Organization of the Dissertation . . . 6

2 Frequent Pattern Mining 8
- 2.0.1 Problem Statement . . . 9
- 2.1 The *Apriori* Algorithm . . . 9
- 2.2 Transaction Layout . . . 11
  - 2.2.1 Horizontal versus Vertical Layout . . . 11
  - 2.2.2 Bitmap Layout . . . 13
- 2.3 Traversal Approaches . . . 14
  - 2.3.1 Breadth-First Versus Depth-First . . . 14
- 2.4 Frequent Itemset Mining . . . 16
  - 2.4.1 All-Patterns: Frequent Pattern Mining Algorithms . . . 18
  - 2.4.2 Closed-Patterns: Frequent Pattern Mining Algorithms . . . 18
  - 2.4.3 Maximal Frequent Mining Algorithms . . . 19
- 2.5 Constraint-based Mining . . . 20
  - 2.5.1 Categories of Constraints . . . 20
  - 2.5.2 Bi-Directional Pushing of Constraints . . . 23
  - 2.5.3 State-of-the-Art Algorithms . . . 25
- 2.6 Parallel Frequent Pattern Mining . . . 26
  - 2.6.1 Replication Algorithms . . . 26
  - 2.6.2 Partitioning Algorithms . . . 28
  - 2.6.3 Hybrid Approach . . . 31
- 2.7 Open-problems and Main Issues . . . 31
  - 2.7.1 Large Data Size . . . 31
  - 2.7.2 High Dimensionality . . . 32
  - 2.7.3 Superfluous Search of the Lattice . . . 33
  - 2.7.4 Observations on Superfluous Processing . . . 33

3 COFI-trees, FP-Tree and the Inverted Matrix 35
- 3.1 Frequent Pattern Tree: Design and Construction . . . 36
  - 3.1.1 Construction of the Frequent Pattern Tree . . . 36
- 3.2 Co-Occurrence Frequent-Item-trees: Construction, Pruning and Mining . . . 37
  - 3.2.1 Construction of the Co-Occurrence Frequent-Item-trees . . . 38
  - 3.2.2 Pruning the COFI-trees . . . 40
  - 3.2.3 Mining the COFI-trees . . . 41
- 3.3 COFI-tree: Experimental Evaluations . . . 43

3.4 Inverted Matrix Layout . . . 45
3.5 COFI-trees Inverted Matrix Based: Design and Construction . . . 49
3.6 Inverted Matrix Algorithm . . . 51
3.6.1 Building the Inverted Matrix . . . 52
3.6.2 Mining the Inverted Matrix . . . 53
3.7 Inverted Matrix: Experimental Evaluations . . . 55
3.8 Summary . . . 56

4 Leap-Traversal approach **59**
4.1 Traversal Approaches . . . 60
4.1.1 Leap-Traversal Approach: Candidate Generation versus Maximal Generation . . . 60
4.1.2 Heuristics used for building and traversing the lexicographic tree . . . 66
4.2 Tree Structures Used . . . 66
4.2.1 Construction of the Headerless Frequent Pattern Tree . . . 67
4.2.2 Construction of the COFI-trees . . . 69
4.2.3 Actual mining of *Frequent-Path-Bases*: The Leap-Traversal approach . 72
4.3 Leap-Traversal Applications . . . 72
4.3.1 Closed and All Frequent Patterns . . . 72
4.4 Performance Evaluations . . . 74
4.4.1 Mining Relatively Small Databases, Real and Synthetic . . . 75
4.4.2 Mining Extremely large synthetic databases . . . 79
4.4.3 Memory Usage . . . 82
4.5 Summary . . . 83

5 Constraint-Based Mining with Leap-Traversal Approach **84**
5.1 Constraints Based Mining . . . 84
5.2 Constraints . . . 86
5.2.1 Bi-directional Pushing of Constraints . . . 86
5.3 Leap with Constraints . . . 87
5.4 Performance Evaluation . . . 90
5.4.1 Impact of $P()$ and $Q()$ selectivity on *BifoldLeap* and Dualminer . . . 92
5.4.2 Scalability tests . . . 92
5.4.3 Constraint checking: pushing constraints versus post-processing . . . 94
5.4.4 Different distributions . . . 95
5.5 Summary . . . 95

6 Parallel Leap-Traversal Approach **98**
6.1 Headerless-Leap versus. COFI-Leap: What to parallelize? . . . 98
6.2 Parallel-Leap-Traversal Approach . . . 99
6.2.1 Load Sharing among Processors . . . 101
6.2.2 Parallel-Leap-Traversal Approach : An Example . . . 102
6.3 Performance Evaluations . . . 104
6.3.1 Effect of Load Distribution Strategy . . . 105
6.3.2 Scalability with respect to the Database Size . . . 105
6.3.3 Scalability with respect to the Number of Processors . . . 106
6.4 Parallel *BifoldLeap* . . . 106
6.4.1 Parallel *BifoldLeap*: An example . . . 108
6.5 Performance Evaluations . . . 109
6.5.1 Scalability with respect to the Database Size . . . 111
6.5.2 Scalability with respect to the Number of Processors . . . 112
6.6 Summary . . . 112

7 Conclusions and Future Work **115**
7.1 Contributions 116
7.2 Future Research 116
7.2.1 Challenges Left to Explore 117
7.2.2 Future Research Trends 119
7.3 Final Thoughts 120

Bibliography **121**

List of Figures

2.1 Itemsets lattice 10
2.2 *Apriori* example 11
2.3 Transactions presented in Horizontal layout 11
2.4 Transactions presented in vertical layout. Grayed cells present transactions IDs for item A 12
2.5 Transactions presented in Bitmap layout 13
2.6 Pattern Lattice with frequent pattern border 14
2.7 Breadth-first traversal 15
2.8 Depth-First traversal 16
2.9 Relation between All, Closed and Maximal patterns 17
2.10 Lattice for all possible itemsets from ABCDE with their respective prices. . . 22
2.11 Prunning of the lattice: CDE violates *monotone* constraints while AB violates *anti-monotone* constraints. All their subsets and supersets are pruned. BDE & AC satisfy the *monotone* & *anti-monotone* constraints respectively. There is no need to check their subsets and supersets. 24

3.1 Steps of phase 1. 36
3.2 Frequent Pattern Tree 37
3.3 COFI-trees 39
3.4 Pruned COFI-trees 41
3.5 Steps needed to generate frequent patterns related to item E 42
3.6 Computational performance and scalability 43
3.7 A: Transactional database. (B): Frequent items circled. (C): Needed Items to be scanned, $\sigma > 4$. 46
3.8 Sub-transactions with items having $\sigma > 4$. (A) List of sub-transactions; (B) Condensed list. 49
3.9 COFI-trees 50
3.10 COFI-trees (A) Item C, (B) Item E 54
3.11 Time needed in seconds to mine different transaction sizes 55
3.12 Time needed in seconds to mine different transaction sizes 56
3.13 Time needed to mine 1M transactions with different supports level 56
3.14 Time needed to mine 1M transactions with different supports levels 57
3.15 Accumulated time needed to mine 1M transactions using four different support levels, including the preprocessing phase 58

4.1 Leap-Traversal 63
4.2 Lexicographic tree of intersections 64
4.3 Headerless FP-Tree: An Example. 67
4.4 Steps needed to generate *Frequent-Path-Bases* from a HFP-Tree 70
4.5 Mining Plant-Protein database 76
4.6 Memory usage while mining Plant-Protein database (support = 30%) 76
4.7 Memory usage while mining Plant-Protein database (support = 15%) 76
4.8 Mining retail database 77

4.9 Memory usage while mining retail database (support = 1%) 78
4.10 Memory usage while mining retail database (support = 5%) 78
4.11 Mining synthetic database. D = 5000 items. Support = 0.5% 79
4.12 Mining synthetic database. D = 5000 items. Support = 0.5% 79
4.13 Mining synthetic database. D = 10000 items. Support = 0.5% 80
4.14 Mining synthetic database. D = 10000 items. Support = 0.5% 80
4.15 Mining extremely large database 81
4.16 Scalability with very large datasets 82
4.17 Memory usage 82

5.1 Pushing $P()$: $Sum(Prices) \leq \$500$ and $Q()$: $Sum(prices) \geq \$100$. 90
5.2 Pushing $P()$, $Q()$, and $P() \wedge Q()$. BifoldLeap $P() \wedge Q()$ is slightly better than BifoldLeap with $Q()$ only despite the fact that $Q()$ is the most selective. With DualMiner Combining $P() \wedge Q()$ is actually not helping. 93
5.3 More selective constraints. 93
5.4 Scalability test 94
5.5 Number of $P()$ and $Q()$ evaluations, using constraint pushing versus post-processing 95
5.6 Effect of changing the price distribution 96

6.1 Parallel Leap-Traversal Steps 100
6.2 Example of Parallel-Leap-Traversal: Finding and intersecting the *Frequent-Path-Bases* 103
6.3 Effect of Leap node distributions 104
6.4 Scalability with respect to the transaction size, (number of processors = 32) . 106
6.5 Scalability with respect to the number of processors, (transaction size = 100M) 107
6.6 Speedup with different support values, (transaction size = 100M) 107
6.7 Example of parallel *BifoldLeap*: finding the *Frequent-Path-Bases* 110
6.8 Intersecting *Frequent-Path-Bases* 110
6.9 Effect of node distributions while pushing both $P()$ and Q constraints. (number of processors = 32) 111
6.10 Scalability with respect to the transaction size while pushing both $P()$ and Q constraints. (number of processors = 32) 112
6.11 Scalability with respect to the number of processors while pushing both $P()$ and Q constraints. (transaction size = 100M) 113
6.12 Speedup with different support values while pushing both $P()$ and Q constraints. (transaction size = 100M) 113

7.1 Transactional database. 117

List of Tables

2.1 Frequent pattern mining algorithms 20
2.2 Commonly used *monotone* and *anti-monotone* constraints 21

3.1 Phase 1, Frequency of each item 47
3.2 Inverted Matrix 47
3.3 scanned Inverted Matrix with $\sigma > 4$ 48
3.4 Example of Sub-transactions with frequent items 49

4.1 Database characteristics 77
4.2 Mining different small datasets: The winner algorithms using high support threshold 77
4.3 Mining different small datasets: The winner algorithms using low support threshold 77

7.1 Compressed Inverted Matrix 118

Acronym List

COFI-Tree : Co-Occurrence Frequent Item Tree.

DM : Data Mining.

FCI : Frequent Closed Itemsets.

FI : Frequent Itemsets.

FIM : Frequent Itemsets Mining.

FMI : Frequent Maximal Itemsets.

FPB : Frequent-Path-Base

FP-Tree : Frequent Pattern Tree.

HFP-Tree : Headerless Frequent Pattern Tree.

IM : Inverted Matrix.

KDD : Knowledge and Data Discovery.

K-Itemset : A set that contains K items.

Publication List

List of Publications from this thesis:

1. Osmar R. Zaïane and Mohammad El-Hajj, Bi-Directional Constraint Pushing in Frequent Pattern mining, Book chapter. Data Mining Patterns: New Methods and Applications, A book to be published by Idea Group Inc, 2007.

2. Mohammad El-Hajj and Osmar R. Zaïane, Parallel Bifold: Large-Scale Parallel Pattern Mining with Constraints, the Journal of Distributed and Parallel Databases. An International Journal, by Springer, 2006.

3. Mohammad El-Hajj and Osmar R. Zaïane, Parallel Leap: Large-Scale Maximal Pattern Mining in a Distributed Environment, International Conference on Parallel and Distributed Systems (ICPADS'06), Minneapolis, MN, USA, July 12-15, 2006, pp 135-142.

4. Mohammad El-Hajj and Osmar R. Zaïane, YAFIMA: Yet Another Frequent Itemset Mining Algorithm, Journal Of digital information management, Volume 3, number 4, pp 243-248, December 2005.

5. Mohammad El-Hajj and Osmar R. Zaïane, Mining with Constraints by Pruning and Avoiding Ineffectual Processing, The 18th Australian Joint Conference on Artificial Intelligence, Sydney, Australia 5-9 December 2005, pp 1001-1004.

6. Mohammad El-Hajj, Osmar R. Zaïane and Paul Nalos, Bifold Constraint-Based Mining by Simultaneous Monotone and Anti-Monotone Checking,, The fifth IEEE International Conference on Data Mining (ICDM'05), Houston, TX, USA, November, 2005, pp 146-153.

7. Osmar R. Zaïane and Mohammad El-Hajj, Pattern Lattice Traversal by Selective Jumps, in Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD), Chicago, IL, USA. August, 2005, pp 729-735.

8. Mohammad El-Hajj, Osmar R. Zaïane, Implementing Leap Traversals of the Itemset Lattice, in ACM SIGKDD Open Source Data Mining Workshop on Frequent Pattern Mining Implementations (OSDM'05), Chicago, IL, USA, August 2005, pp 16-25.

9. Mohammad El-Hajj, Osmar R. Zaïane Finding All Frequent Patterns Starting From The Closure, The First International Conference on Advanced Data Mining and Applications Wuhan, China, July, 2005, pp 67-74.

10. Mohammad El-Hajj, Osmar R. Zaïane On Frequent Itemset Mining with Closure, 2nd International Conference on Information Technology (ICIT 2005), Amman, Jordan May, 2005. pp 21-30.

11. Osmar R. Zaïane, Mohammad El-Hajj, Yi Li, Stella Luk, Scrutinizing Frequent Pattern Discovery Performance, IEEE ICDE, Tokyo, Japan, April, 2005, pp 1109-1110.

12. Mohammad El-Hajj and Osmar R. Zaïane, COFI Approach for Mining Frequent Itemsets Revisited, 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD-04), Paris, France, June 2004.

13. Osmar R. Zaïane and Mohammad El-Hajj, Advances and Issues in Frequent Pattern Mining, Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04), Sydney, Australia, May, 2004. (Conference Tutorial Notes)

14. Mohammad El-Hajj and Osmar R. Zaïane, COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation, in Workshop on Frequent Itemset Mining Implementations (FIMI'03) in conjunction with IEEE-ICDM 2003, Melbourne, Florida, USA, 2003.

15. Mohammad El-Hajj and Osmar R. Zaïane, Inverted Matrix: Efficient Discovery of Frequent Items in Large Datasets in the Context of Interactive Mining, in Proc. 2003 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD), Washington, DC, USA, August, 2003, pp 109-118.

16. Mohammad El-Hajj and Osmar R. Zaïane, Non Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations, in Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003), Prague, Czech Republic, September, 2003, pp 371-380.

17. Mohammad El-Hajj and Osmar R. Zaïane, Parallel Association Rule Mining with Minimum Inter-Processor Communication, Fifth International Workshop on Parallel and Distributed Databases (PaDD'2003) in conjunction with the 14th Int' Conf. on Database and Expert Systems Applications DEXA2003, Prague, Czech Republic, September, 2003, pp 519-523.

18. Osmar R. Zaïane, Mohammad El-Hajj, Paul Lu, Fast Parallel Association Rule Mining Without Candidacy Generation, in Proc. of the IEEE 2001 International Conference on Data Mining (ICDM'2001), San Jose, CA, USA, November, 2001, pp 665-668.

Chapter 1

Introduction

The beginning is the half of every action.
– Greek Proverb

The last decades have witnessed a massive growth in data collection techniques from different resources like satellite images, surveillance cameras, and commercial domain transactions; this has led to a huge archiving of data without the ability to extract all the needed and useful information. Moreover, increasing the market competition motivates the need to maintain the customer and to improve the ability to compete. Due to this competition, companies started to realize the need for analyzing their data repositories for decision support reasons. They also realized that automation of the knowledge discovery process has become vital to achieve this target.

Data mining [79] is a major step in the process in which concealed knowledge and correlations can be extracted from a store of databases or facts.

One of the canonical tasks in data mining is the discovery of association rules. It attempts to find the relation between different items and how often they may occur together in the same transaction. An example can be taken from the retail industry where we can study the relation between items; with the purpose of discovering the customer habits, to make some decision regarding the sales times for such items or even the way items should be ordered in the store display. The association-mining task can be broken into two steps:

- A major step for finding all Frequent Itemsets (FI) that occurs together while satisfying a minimum threshold.
- A straightforward step for generating strong rules from the found frequent itemsets.

1.1 Motivation

Despite the importance and benefits of Frequent Itemsets, and the development of many efficient frequent-pattern mining techniques, and the plethora of research done on this area, ma-

jor problems still persist. Most of the existing techniques assume relatively small databases, leaving extremely large but realistic datasets out of reach. Databases exist that are in the order of hundreds of millions of transactions and thousands of items such as those for big stores and companies similar to WalMart, UPS, etc. With the billions of radio-frequency identification chips (RFID) expected to be used to track and access every single product sold in the market[1], the sizes of transactional databases will be overwhelming even for current state-of-the-art algorithms.

Existing association rule mining algorithms suffer from many problems when mining massive transactional datasets. One major problem is the high memory dependency: either the gigantic data structure built is assumed to fit in main memory, or the recursive mining process is too voracious in memory resources. Another major impediment is the repetitive and interactive nature of any knowledge discovery process. To tune mining parameters, many runs of the same algorithms are necessary leading to the building of huge data structures for each mining iterations. A third problem is related to a set of algorithms that require multiple full I/O scans of the database, which makes it infeasible to mine extremely large databases.

1.2 Problem Statement

The problem of mining frequent itemsets stems from the problem of mining association rules over market basket analysis as introduced by Agrawal et al. [2]. The problem consists of finding sets of items (i.e. itemsets) that are sufficiently frequent in a transactional database. The data could be retail sales in the form of customer transactions, text documents [39], or even medical images [86]. These frequent itemsets have been shown to be useful for other applications such as recommender systems [57], diagnosis [41], decision support [23], telecommunication [54], and even supervised classification [58, 4]. They are used in inductive databases [60], query expansion [72], document clustering [9], etc. Lately, new studies show that mining for the set of all frequent itemsets could be infeasible in many situations and new ideas have emerged to find only the non redundant patterns, such as the closed patterns. In other cases, finding only the maximal subsets of these patterns could be the answer. Even injecting constraints into the set of frequent patterns to find only the set of real interest to decision makers could be the solution. Formally, as defined by Agrawal et al. [3], the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items and m is considered the dimensionality of the problem. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. An itemset X is said to be *frequent* if its *support* s (i.e. the ratio

[1] Rick Whiting, Data Avalanche, InformationWeek, Feb. 16, 2004. http://www.informationweek.com/story/showArticle.jhtml?articleID=17700027

of transactions in $\mathcal{D}$ that contain X is greater than or equal to a given minimum support threshold σ). A frequent itemset $\mathcal{M}$ is considered maximal if there is no other frequent set that is a superset of $\mathcal{M}$. A frequent itemset $\mathcal{C}$ is considered closed if there is no other frequent set that is a superset of $\mathcal{C}$ and has the same support as $\mathcal{C}$.

1.3 Thesis Statement

In this work, we investigate the possible issues that prevent existing state-of-the-art algorithms from mining extremely large databases. The central thesis statement of this work is presented as follows:

> *A framework of sequential and parallel algorithms can be implemented to mine extremely large databases by applying two strategies which are: First, by using a novel traversal searching for frequent patterns while using minimal memory requirements. Second, by designing algorithms specifically made for parallel execution rather than parallelizing an algorithm designed for a sequential execution.*

This research work justifies empirically and practically that mining extremely large databases is possible by applying new techniques. The major issues addressed to support the thesis statements of this research are as follows:

- Investigate having new database layouts, other than the traditional horizontal versus vertical formats.
- Investigate building small memory structures that facilitate mining frequent patterns without causing major memory load.
- Investigate having new traversal search approach for frequent patterns other than the traditional top-down versus bottom-up approaches.
- Investigate the suitability of the suggested approach with other types of frequent items such as closed or maximal patterns.
- Investigate enhancing this approach with constraints efficiently.
- Investigate parallelizing this approach, mainly when mining extremely large databases and minimizing communication costs between processors while optimizing load balance.

1.4 Thesis Contributions

The major contribution of this thesis can be summarized as follows:

1. **COFI-trees**: The (Co-Occurrence Frequent Item Tree, or COFI-tree for short) The COFI-tree approach is a divide-and-conquer approach, in which we do not seek to find all frequent patterns at once, but we independently find all frequent patterns related to each frequent item in the frequent 1-itemset.

   The main advantages of using COFI-trees are:

   - We only build one COFI-tree for each frequent item A. This COFI-tree is non recursively traversed to generate all frequent patterns related to item A.
   - Only one COFI-tree resides in memory at one time and it is discarded as soon as it is mined to make room for the next COFI-tree. This idea is described in Chapter 3

2. **Inverted Matrix**: A novel layout that prevents multiple scanning of the database during the mining phase, in which finding frequent patterns could be achieved in less than a full scan with random access. This layout supports interactive mining where tuning parameters can be applied for many runs of the same algorithm without the need to rebuild huge data structures again. Inverted Matrix is explained in Chapter 3.

3. **Leap-Traversal**: The strategy used to find frequent patterns always includes the traversal of the lattice of candidate patterns. Leap-Traversal is a new traversal approach that jumps in the search space among only promising nodes and avoids nodes that would not participate in the answer set and drastically reduces the number of candidate patterns. This approach finds efficiently the set of all, closed and maximal patterns using special motifs which can be used to generate the support for all tested pattern. Moreover, It depicts efficient performance in pushing constraints in duality. Chapters 4 and 5 present the results for the Leap approach.

4. **Parallel Leap**: A new parallel Leap is being proposed in this research, where database sizes never reported in the literature before has been mined efficiently. Different types of frequent patterns have been generated in parallel. Constraints pushing is also shown to be efficiently implemented. Chapter 6 presents the results of this parallel approach.

1.5 Research Methodology

This research was conducted in a phased approach, as follows:

Evaluation of prior work: Research in the area of Frequent Itemset Mining (FIM) started in the last decade with a wide range of efficient algorithms. We surveyed existing algorithms and identified issues that prevented existing algorithms from mining extremely

large databases. The existing algorithms and issues with FIM are reviewed in Chapter 2.

Conception and design of new FIM Framework: We designed new methods to address the problem of FIM. These methods cover the database layout, traversal strategy, and the suitability of generating different types of patterns, and the capability of parallelizing these methods.

Implementation and test: We implemented our new set of algorithms against the state-of-the-art algorithms provided by their authors. We applied our methods to real datasets to test their effectiveness, and to extremely large synthetic datasets to study their performance and scalability.

Evaluation: We compared and evaluated our work with the published results. We evaluated the effectiveness and scalability of our algorithms as well.

Dissemination: The results of this research are disseminated through submission of papers to peer reviewed conferences and journals [18, 24, 25, 26, 27, 28, 29, 30, 31, 32, 32, 33, 34, 35, 36, 37, 82, 83, 84, 85]. The review process and discussions at conferences were essential for further improvement of our research work.

1.6 Organization of the Dissertation

The rest of this dissertation is organized as follows:

- In Chapter 2, we present the state-of-the-art work in the area of frequent pattern mining in both paradigms: sequential and parallel. We highlight the existing algorithms, database layouts used by these algorithms, traversal strategies, and types of frequent patterns generated. In this chapter we also discuss the main issues and open problems in this field of research.

- In Chapter 3, we present novel small memory structures that we use for the mining process. Using these data structures we show how we can generate the set of all frequent patterns using minimal memory structures. A new database layout is also presented in this chapter called Inverted Matrix that supports interactive mining, and could in many cases mine for frequent pattern in less than 1 full I/O scan.

- In Chapter 4, we present a novel traversal strategy that jumps in the search space among only promising nodes. Our leaping approach avoids nodes that would not participate in the answer set and drastically reduces the number of candidate patterns. We use this approach to efficiently pinpoint maximal patterns at the border of the

frequent patterns in the lattice and collect enough information in the process to generate all subsequent patterns. We also show that this strategy can be used to generate all types of patterns (all, closed, and maximals) by making use of novel patterns that facilitate the enumerations by counting of any pattern without having to revisit the database.

- In Chapter 5, we show how mining for frequent itemsets can generate an overwhelming number of patterns, often exceeding the size of the original transactional database. To solve this issue we propose an approach that allows the efficient mining of frequent itemset patterns, while considering constraints early in the process.

- In Chapter 6, we show the applicability of the Leap approach for parallelization, and how by parallelizing this approach we could mine in reasonable time extremely large database. We also show that different types of frequent patterns can be generated in parallel, and how constraints can be pushed in this environment.

- In Chapter 7, we conclude this dissertation with a summary of our contributions in the frequent itemset mining research area. It also draws a path for a future direction as a continuation of this work.

Chapter 2

Frequent Pattern Mining

Review your goals twice every day in order to be focused on achieving them.
– Les Brown

The advances in data collection methods and the huge amount of data being stored nowadays prompted the existence of the field of knowledge discovery and data mining (KDD). One of the core tasks of this field is the frequent itemset mining with thorough applications ranging from mining association rules [2], bioinformatics [22], security [71], correlations [13], classifications [56], and medical images [49] where mining for association rules is still the most common one among them. The real motivation for mining association rules is to study the customer purchase habits as they describe how often items are bought together in one transaction.

Since the introduction of frequent itemset mining [2], many algorithms have been proposed to solve it. Where they differ is in their strategies in traversing the search space, the number of scans of the database, and their required database format. Lately, new studies show that mining for the set of all frequent patterns could be infeasible in many situations and new ideas have emerged to find only the non redundant patterns, such as the closed patterns. In other cases, finding only the maximal subsets of these patterns could be the answer. Even injecting constraints into the set of frequent patterns to find only the set of real interest to decision makers could be the solution.

Many studies and workshops have been conducted to identify the real problems that face this field of research, and how researchers can deal with these issues. In this chapter, we will try to highlight the main strategies used for frequent pattern mining, in terms of database formats, traversal methods, forms of frequent patterns that can be found, and the most influential state-of-the-art algorithms in this area. Finally, we draw attention to the main issues and open problems.

2.0.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [2]. The problem consists of finding associations between items or itemsets in transactional data. The data is typically retail sales in the form of customer transactions, but can be any data that can be modeled into transactions. For example, medical images where each image is modeled by a transaction of visual features from the image [86], or text data where each document is modeled by a transaction representing a bag of words [39], or web access data where click-stream visitation is modeled by sets of transactions [75], all are well suited applications for association rules or frequent itemsets. Association rules have been shown to be useful for other applications such as recommender systems [57], diagnosis [41], decision support [23], telecommunication [54], and even supervised classification [58, 4]. The main and most expensive component in mining association rules is the mining of frequent itemsets, which has been described in the previous chapter.

Discovering association rules, however, is nothing more than an application for frequent itemset mining, like inductive databases [60], query expansion [72], document clustering [9], etc. What is relevant here is the efficient counting of some specific itemsets.

This chapter starts with the description of the frequent pattern mining problem with a brief explanation of the *Apriori* algorithm, which is one of the most influential algorithms in this research area. The database format used by the mining algorithms is depicted in Section 2.2, and the traversal strategies are illustrated in Section 2.3. Generating the set of all patterns is not feasible in some cases, mainly while mining either extremely large databases or extremely dense one. Many solutions are proposed in the literature to lessen the effect of such database either by finding sample presentations of the frequent patterns, by finding the non redundant closed or maximal patterns, or by applying constraints on the mining patterns to generate patterns that are of more interest to the users or even by parallelizing the mining process. Section 2.4 presents the types of frequent mining methods with their state-of-the-art algorithms. Section 2.5 presents the two main types of constraints and how they are implemented. Section 2.6 presents the current methods taken toward parallelizing the frequent pattern mining approach. Finally, we discuss in Section 2.7 the open problems and main issues that motivate our work.

2.1 The *Apriori* Algorithm

The problem of mining frequent patterns was introduced by Agrawal et al. [2]. Shortly after that a novel algorithm called *Apriori* was introduced by Agrawal et al. [3]. *Apriori* is considered now to be one of the key algorithms, and seems to be the most popular in many applications for enumerating frequent itemsets. This *Apriori* algorithm also forms

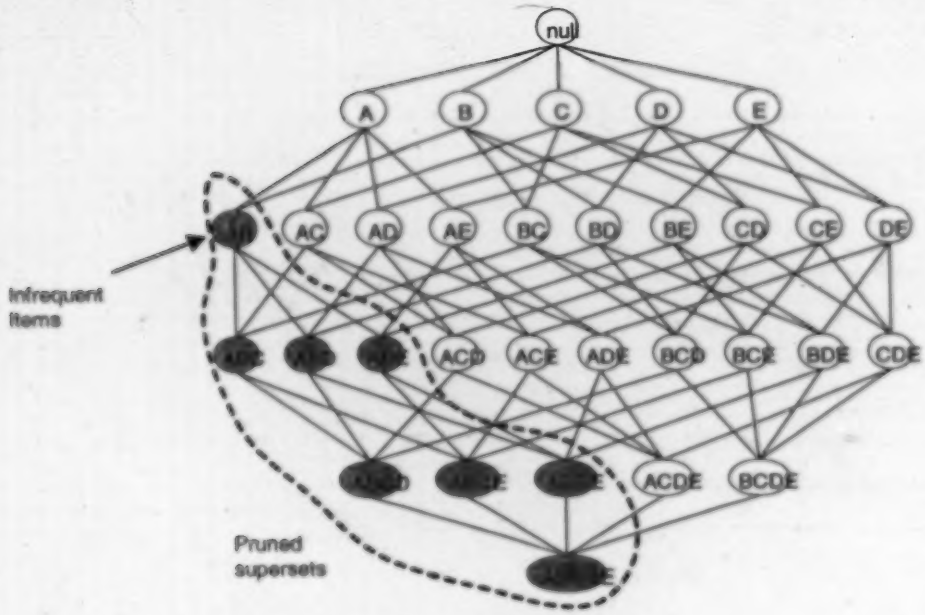


Figure 2.1: Itemsets lattice

the foundation of most known algorithms. A set consisting of k items is called a k-itemset. Given a k-itemset α, it defined $k-1$ subsets, each containing $k-1$ items. Each subset is formed by removing one item from α. These subsets are the $k-1$ itemsets of α. *Apriori* uses a *monotone* property stating that ***for a k-itemset to be frequent, all its (k-1)-itemsets have to be frequent.*** The use of this fundamental property reduces the computational cost of candidate frequent itemset generation. Figure 2.1 shows that if a single non frequent pattern such as AB is found, then all its supersets must also be non frequent and consequently could be pruned. Thus large areas of the search space could be avoided.

The *Apriori* algorithm starts its first scan with the goal of counting all items. The set of counted items are called 1-itemset candidates where all non frequent items are removed and only frequent pairs are used to find the set of 2-itemset candidates. The support for each of these candidates is computed by another scan of the database. Non frequent candidates are removed and only frequent ones are used to generate the candidate of size 3, and so on until no more candidate items can be generated.

Although *Apriori* algorithm can efficiently prune the search space, it uses many database scans that cause real issues while mining large datasets. It also suffers from high computational cost of finding the set of candidate items. Figure 2.2 presents a full example of the *Apriori* algorithm for a sample database of 4 transactions containing 5 items with support threshold greater than 1.

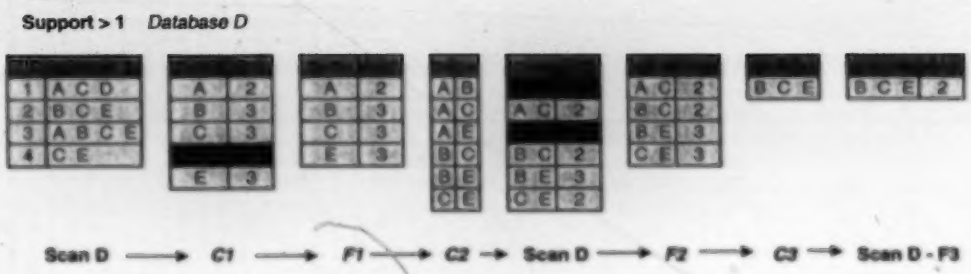


Figure 2.2: *Apriori* example

2.2 Transaction Layout

The transaction layout is the method in which items in transactions are formatted in the database. Currently, there are two main approaches: the horizontal approach [50] and the vertical one [87]. Other approaches have also been introduced as variations of those two main approaches such as the bitmap approach [6].

Transaction ID	Items
1	A G D C B
2	B C H E D
3	B D E A M
4	C E F A N
5	A B N O P
6	A C Q R G
7	A C H I G
8	L E F K B
9	A F M N O
10	C F P J R
11	A D B H I
12	D E B K L
13	M D C G O
14	C F P Q J
15	B D E F I
16	J E B A D
17	A K E F C
18	C D L B A

Figure 2.3: Transactions presented in Horizontal layout

2.2.1 Horizontal versus Vertical Layout

The relational database model consists of storing data into two-dimensional arrays called tables. Each table is made of N rows called features or observations, and M columns called attributes or representing variables. The format of storing transactions in the database plays an important role in determining the efficiency of the association-rule-mining algorithm used.

Most of the existing algorithms use one of the two layouts, namely horizontal and vertical. The horizontal layout, which is the most commonly used, relates all items on the same transaction together. In this approach the ID of the transaction plays the role of the key for the transactional table. Figure 2.3 represents a sample of 18 transactions made of items from A to R. The number of items in a transaction need not to be the same.

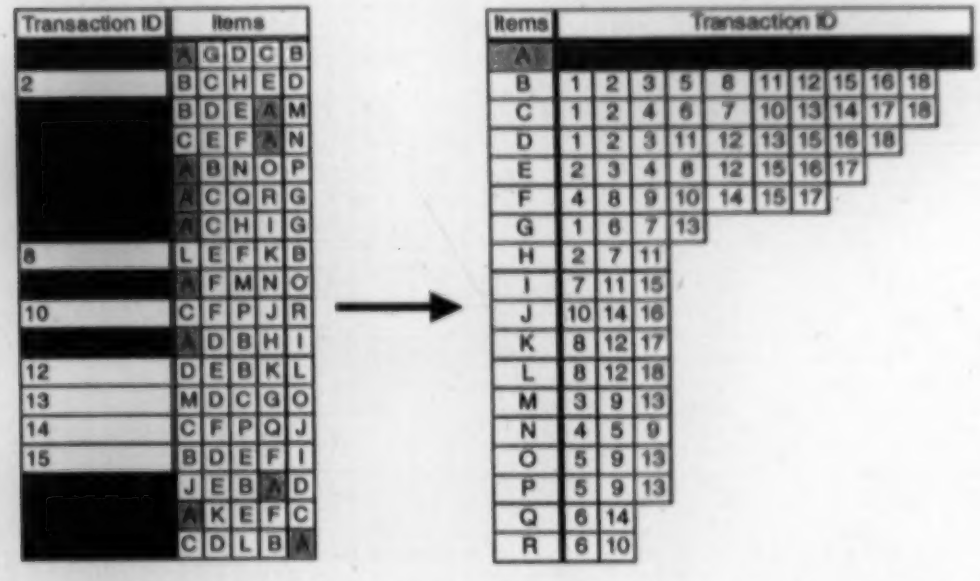


Figure 2.4: Transactions presented in vertical layout. Grayed cells present transactions IDs for item A

The vertical layout relates all transactions that share the same items together. In this approach the key of each record is the item. Each record in this approach has an item with all transaction numbers in which this item occurs. This is analogous to the idea of inverted index [7] in information retrieval, where a word is associated with the set of documents it appears in. Here the word is an item and the document is a transaction. Transactions in Figure 2.3 are presented by using the vertical approach in Figure 2.4. The horizontal layout has a very important advantage: it combines all items in one transaction together where by using some clever techniques, such as the one used by [48], the candidacy generation step can be eliminated. On the other hand, this layout suffers from some limitations such as superfluous processing (which will be discussed later in this chapter) since there is no index to the items. The vertical layout, however, is an index on the items in itself and reduces the effect of large data sizes as there is no need to always re-scan the whole database. On the other hand, this vertical layout still needs the expensive candidacy generation phase. Also computing the frequencies of itemsets becomes a tedious task of intersecting records of different items of the candidate patterns. In [87] a vertical database layout is combined with

clustering techniques and hypergraph structures to find frequent itemsets. The candidacy generation and the additional steps associated with this layout make it impractical to mine extremely large databases. Both approaches show good performance for cases of updating the transactional databases. The horizontal layout databases are updated simply by adding new records (lines) to the database, while the vertical layout is updated by either adding new items to the index for the case of introducing new items or simply adding new transaction numbers to the item list. Most of the state-of-art frequent pattern mining algorithms, such as *Apriori* [3], *FP-Growth* [50], and CLOSET+ [78], use the horizontal layout. The vertical approach is used by other algorithms such as Eclat [87], GenMax [44], and CHARM [88]. Detailed explanations of these algorithms are given later in this chapter.

T#	Items				
T2	B	C	H	E	D
T3	B	D	E	A	M
T4	C	E	F	A	N
T5	A	B	N	O	P
T6	A	C	Q	R	G
T7	A	C	H	I	G
T8	L	E	F	K	B
T9	A	F	M	N	O
T10	C	F	P	J	R
T11	A	D	B	H	I
T12	D	E	B	K	L
T13	M	D	C	G	O
T14	C	F	P	Q	J
T15	B	D	E	F	I
T16	J	E	B	A	D
T17	A	K	E	F	C
T18	C	D	L	B	A

Transaction ID	Items																	
T#	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T2	0	1	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
T3	1	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0
T4	1	0	1	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0
T5	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
T6	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
T7	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
T8	0	1	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0
T9	1	0	0	0	0	1	0	0	0	0	0	0	1	1	1	0	0	0
T10	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	0	1
T11	1	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
T12	0	1	0	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0
T13	0	0	1	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0
T14	0	0	1	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0
T15	0	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
T16	1	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0
T17	1	0	1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0
T18	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Figure 2.5: Transactions presented in Bitmap layout

2.2.2 Bitmap Layout

The bitmap layout [6] is a variation of the horizontal approach where it can be viewed as a matrix in which rows present the transactions and columns present the items. If an item in column x exists in transaction y then the cell (x,y) in the matrix is set to true otherwise it is set to false. This approach is suitable when the dimension of the problem is relatively small because the number of columns becomes small and maintaining this matrix is manageable. Figure 2.5 presents a transactional database and its presentation in the bitmap approach. The main advantage of this layout is the efficient counting of support. Finding the support of any candidate pattern is simply applying the bitwise AND of the

candidate pattern against each record to detect if the candidate pattern is a subset of the record or not, and consequently incrementing its support count by 1 or not. MAFIA [16] is an example of an algorithm that uses a bitmap approach.

2.3 Traversal Approaches

The search space for $|I|$ frequent itemset contains exactly $2^{|I|}$ different itemsets as in Figure 2.1. If I is large enough, then the naïve approach to generate and count the supports of all itemsets over the database cannot be achieved within a reasonable period of time. This motivates many pruning ideas that have been introduced in the literature. The way of traversing the search space is an important issue for frequent pattern mining algorithms. Most of the existing frequent pattern mining algorithms use either breadth-first-search [2] or depth-first-search [50] strategies to find candidates that will be used to determine the frequent patterns. Other approaches have been proposed such as a hybrid one that combines both the depth and breadth approaches.

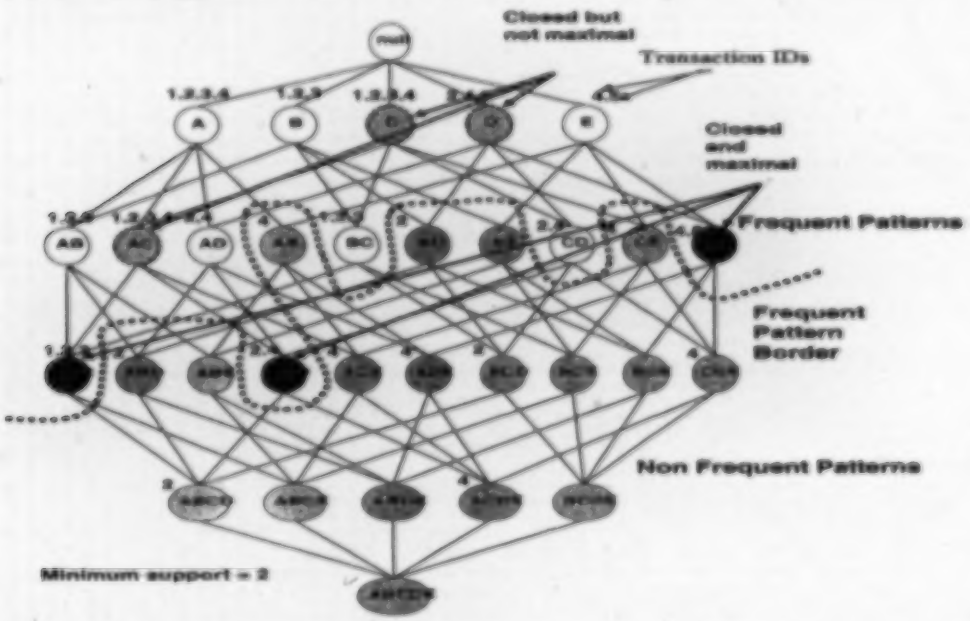


Figure 2.6: Pattern Lattice with frequent pattern border

2.3.1 Breadth-First Versus Depth-First

Assume a lattice made of 5 frequent items contained in 5 transactions as in Figure 2.6. To mine this lattice with a minimum support equal to 2, we need to define a *Frequent Pattern Border*. *Frequent Pattern Border* is the border that separates frequent items that need to be discovered from the non frequent ones. This border is illustrated in Figure 2.6.

To identify the frequent patterns using the breadth-first-search we need to traverse the lattice level-by-level: where this approach uses knowledge of frequent patterns at level k to generate candidates at level $k+1$ before omitting the non frequent ones and keeping the frequent ones to be used for the level $k+2$ candidates, and so on. This approach usually uses many database scans, and it is not favored while mining databases that are made of long frequent patterns, i.e frequent k itemset with a large k. Figure 2.7 presents the same lattice as in Figure 2.6 traversed breadth-first, where the frequent 1-itemsets are first generated, then used to generate longer candidates to be tested from size two and above.

In our token example, in Figure 2.7, this approach would test 18 candidates to finally discover the 13 frequent ones assuming that the support threshold is equal to 2. Five were unnecessarily tested, since they ended-up infrequent. Conversely, depth-first-search tries to detect the long patterns at the beginning and only back-tracks to generate the frequent patterns from the long ones that are already been declared as frequent. For longer patterns, depth-first-search outperforms the breadth-first method. But in cases of sparse databases where many long candidates do not occur frequently, then the depth-first-search is shown to have poor performance. The same example in Figure 2.7 is presented in Figure 2.8 using the depth-first approach. In this case 23 candidates were tested, 10 unnecessarily.

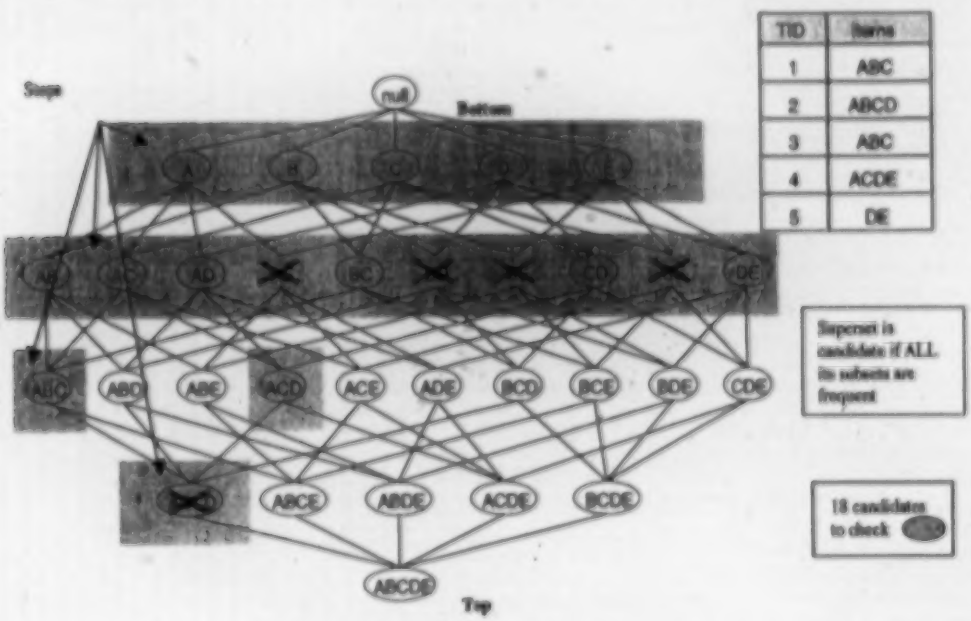


Figure 2.7: Breadth-first traversal

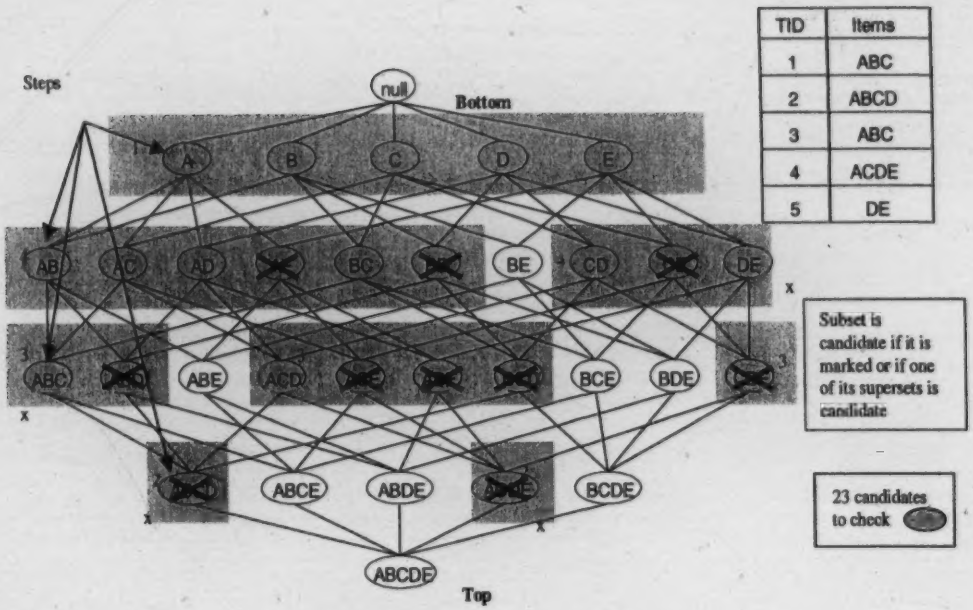


Figure 2.8: Depth-First traversal

2.4 Frequent Itemset Mining

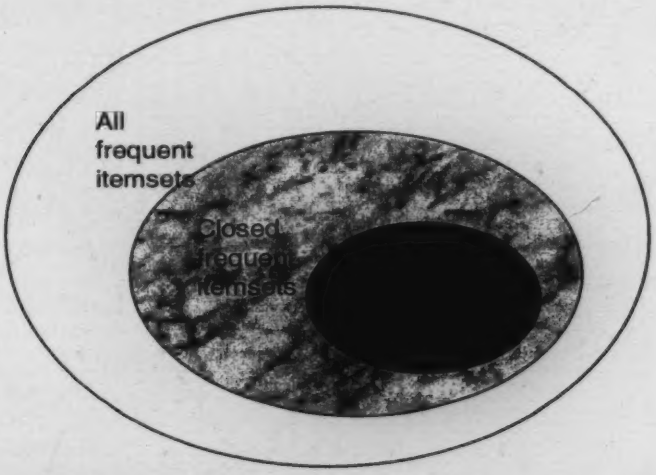
Discovering frequent patterns is a fundamental problem in data mining. Many efficient algorithms have been published on this problem in the last 10 years. Most of the existing methods operate on relatively small databases. Given different small datasets with different characteristics, it is difficult to say which approach would be a winner. Moreover, on the same dataset with different support thresholds, different winners could be proclaimed. Difference in performance becomes clear only when dealing with very large datasets. Novel algorithms, otherwise victorious with small and medium datasets, can perform poorly with extremely large datasets. One of the questions that the frequent mining community asks is: Is it possible to mine efficiently for frequent itemsets in extremely large transactional databases? Databases either in the order of millions of transactions and thousands of items such as those for big stores and companies like WalMart, UPS, etc. There is obviously a chasm between what we can mine today and what needs to be mined. It is true that new attempts toward solving such problems are made by finding the set of frequent closed itemsets (FCI) [65, 78, 88] and the set of maximal frequent patterns [1, 8, 16, 44, 45].

Definition 2.1 (*Closed Patterns*)
A frequent itemset X is closed if and only if there is no X' such that $X \subseteq X'$ and the support of X equals to the support of X'. □

Definition 2.2 (*Maximal Patterns*)

A frequent itemset X is said to be maximal if there is no frequent itemset X' such that $X \subseteq X'$. □

Frequent maximal patterns are a subset of frequent closed patterns, which are a subset of all frequent patterns, as in Figure 2.9. Finding only the closed item patterns dramatically reduces the size of the results set without loosing relevant information. Closed itemsets reduce the redundancy already in the set of all frequent itemsets. From the closed itemsets one can derive all frequent itemsets and their counts. Directly discovering or enumerating closed itemsets can lead to huge time saving during the mining process.



Maximal frequent itemsets ⊆ Closed frequent itemsets ⊆All frequent itemset

Figure 2.9: Relation between All, Closed and Maximal patterns

The set of maximal frequent itemsets is generally found to be orders of magnitude smaller in size than the set of closed itemsets, and the set of closed itemsets is generally found to be orders of magnitude smaller in size than the set of all frequent itemsets. While we can derive the set of all frequent itemsets directly from the maximal patterns, their support cannot be obtained without counting. Nonetheless, discovering maximal patterns has interesting significance, especially in pattern clustering applications where frequent patterns are important, not their exact support. These attempts toward discovering only the set of closed or maximal patterns have also not been tested in the literature toward mining extremely large datasets.

. While there are myriad algorithms to discover all frequent, the closed, and maximal patterns, their performances are indistinguishable for small and medium size databases.

Experimental results are typically reported with few hundred thousand transactions. A recent study by Zheng et al. [90] has even shown that with real datasets, *Apriori*, the oldest algorithm for mining frequent itemsets, outperforms the newer approaches. Moreover, when results are discovered in few seconds, performance becomes almost irrelevant. The problem of performance becomes a real issue when the size of the database increases significantly (in the order of millions of transactions) or when the dimensionality of the problem increases (i.e. the number of distinct items in the database).

2.4.1 All-Patterns: Frequent Pattern Mining Algorithms

There is a plethora of algorithms proposed in the literature to address the issue of discovering frequent itemsets and generating association rules. Some of these algorithms are:

- *Apriori* [3] is the most important, and at the basis of many other approaches. This algorithm has been discussed earlier in this chapter.
- *ECLAT* [87] is the first known depth-first algorithm that uses the vertical format. It requires only one full database scan to build the vertical database. Then a set of intersections is implemented to find the set of all frequent patterns. One of the main advantages of this algorithm is that it has a very fast support counting mechanism. However, it suffers once the intermediate lists of items with their list of transaction IDs become larger than the memory.
- *FP-Growth* [50] is another approach that avoids generating and testing many itemsets. *FP-Growth* generates, after only two I/O scans, a compact prefix tree called FP-Tree representing all sub-transactions with only frequent items. A clever and elegant recursive method mines the tree by creating projections called conditional trees and discovers patterns of all lengths without directly generating all candidates the way Apriori does. However, the recursive method to mine the FP-Tree requires significant memory, and large databases quickly blow out the memory stack. *FP-Growth* requires horizontal format.

For other frequent pattern mining algorithms the reader may refer to good surveys at [42] and [52]

2.4.2 Closed-Patterns: Frequent Pattern Mining Algorithms

Finding the set of closed frequent patterns has been studied in the literature [80]. In this section we will discuss only four state-of-the-art algorithms in this area:

- *A-Close* [65] is an *Apriori*-like algorithm. This algorithm mines directly for closed frequent itemsets. It uses a breadth-first search strategy. This algorithm is one-order of magnitude faster than *Apriori*, when mining with a small support. This

algorithm shows, however, poor performance compared to *Apriori* when mining with high support especially when we find a small set of frequent patterns as it consumes most of its computation power in computing the closure of itemsets. This algorithm also shows weak results when mining relatively long patterns.

- *CLOSET+* [78] is an extension of the *FP-Growth* algorithm. It builds recursively conditional trees that cause this algorithm to suffer when mining for low support threshold. This algorithm reacts based on the sparsity of the database as it uses a 2-level hash index result tree structure for a dense database and uses pseudo projection-based upward-checking for a sparse database.

- *MAFIA* [16] is originally designed to mine for maximal itemsets, but it has an option to mine for closed itemsets. It uses a vertical bitmap representation.

- *CHARM* [88] is similar to ECLAT in using a vertical representation of the database. It adopts the *diffset* [89] technique to reduce the size of intermediate transaction IDs.

2.4.3 Maximal Frequent Mining Algorithms

Mining for the set of maximal patterns has also been investigated at length. Some of the state-of-the–art algorithms in this area are:

- *MaxMiner* [8] is an *Apriori*-like algorithm that in some cases needs to scan the database k times to find a pattern of length k. This algorithm performs a breadth-first traversal of the search space. At the same time it performs intelligent pruning techniques to eliminate irrelevant paths from the search tree. A look-ahead strategy achieves this, where there is no need to further process a node if it, with all its extensions, is determined to be frequent. To improve the effectiveness of the superset frequency pruning, MaxMiner uses a reorder strategy.

- *DepthProject* [1], performs a depth-first search of the lexicographic tree of the itemsets with some superset pruning. It also uses a look-ahead pruning with item reordering. The result of the mining process of *DepthProject* is a superset of maximal patterns and requires a post-pruning to remove non maximal patterns.

- *MAFIA* [16], which is one of the fastest maximal algorithms, uses many pruning techniques such as the look-ahead used by the *MaxMiner*, which is checking if a new set is subsumed in another existing maximal set, and other clever heuristics.

- *GENMAX* [44] is a vertical approach that uses a novel strategy, progressive focusing, for finding supersets. In addition, it counts supports faster using diffsets [89].

- *FPMAX* [45], is an extension of the *FP-Growth* method, for mining the set of maximal patterns. This algorithm creates beside the FP-Tree and conditional trees another trie structure called a maximal frequent item tree to store all maximal frequent items. *FPMAX* received the FIMI best implementation award for 2003 [43].

Table 2.1 presents a comparison between some of the latest state-of-the-art algorithms in terms of transaction layout, search space, number of scans the algorithms need, and the target frequent pattern type.

Table 2.1: Frequent pattern mining algorithms

Algorithm	Target Output	No. of database scans	Traversal	Layout
Apriori	All	n	BFS	Horizontal
ECLAT	All	best case 2	DFS	Vertical
FP-Growth	All	2	DFS	Horizontal
A-Close	Closed	n	BFS	Horizontal
CLOSET+	Closed	2	DFS	Horizontal
CHARM	Closed	best case 2	DFS	Vertical
MaxMiner	Maximal	n	BFS	Horizontal
DepthProject	Maximal	2	DFS	Horizontal
MAFIA	Maximal	2	DFS	Bitmap
GENMAX	Maximal	best case 2	DFS	Vertical
FPMAX	Maximal	2	DFS	Horizontal

2.5 Constraint-based Mining

It is known that algorithms for discovering association rules generate an overwhelming number of those rules. While many new very efficient algorithms were recently proposed to allow the mining of extremely large datasets, the problem of the sheer number of rules discovered still remains. The set of discovered rules is often so large that it becomes useless. Different measures of interestingness and filters have been proposed to reduce the discovered rules, but one of the most realistic ways to find only those interesting patterns is to express constraints on the rules we want to discover. However, filtering the rules post-mining adds a significant overhead. Ideally, dealing with the constraints should be done during the mining process as early as possible. In general, two types of constraints – ***monotone*** and ***anti-monotone*** – have been identified.

2.5.1 Categories of Constraints

Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A unique identifier *TID* is given to each transaction. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. An itemset

X is said to be *large* or *frequent* if its *support* s is greater than or equal to a given minimum support threshold σ. In addition to the transactions, other tables could describe the items in the transactions in terms of attributes such as price, weight, size, etc. A constraint ζ is a predicate on itemset X that yields either *true or false* and is typically expressed in terms of the items' attributes. An itemset X satisfies a constraint ζ if and only if $\zeta(X)$ is *true*. For example: $Price(X) \leq \$100$. Many constraints can be associated with frequent patterns mining. In this work we discuss two common types of constraints which are *anti-monotone* and *monotone*.

Definition 2.3 (*Anti-monotone constraints*)
A constraint ζ is *anti-monotone* if and only if when an itemset X violates ζ, then any superset of X also violates ζ. In other words, if ζ holds for an itemset S then it holds for any subset of S. □

Many constraints fall within the *anti-monotone* constraints category. The minimum support threshold is a typical *anti-monotone* constraint. For example, the $sum(S) \leq v(\forall a \in S, a \geq 0, v \geq 0)$ is an *anti-monotone* constraint. Assume that items A, B, and C have prices \$100, \$150, and \$200 respectively. Given the constraint ζ: $(sum(S) \leq \$200)$, then since itemset AB with a total price of \$250 violates the ζ constraint, there is no need to test any of its supersets such as ABC as they also violate the constraint ζ.

Table 2.2: Commonly used *monotone* and *anti-monotone* constraints

monotone	*anti-monotone*
$min(S) \leq v$	$min(S) \geq v$
$max(S) \geq v$	$max(S) \leq v$
$count(S) \leq v$	$count(S) \geq v$
$sum(S) \geq v(\forall a \in S, a \geq 0)$	$sum(S) \geq v(\forall a \in S, a \leq 0)$
$range(S) \geq v$	$range(S) \leq v$
$support(S) \leq v$	$support(S) \geq v$

Definition 2.4 (*Monotone constraints*)
A constraint ζ is *monotone* if and only if when an itemset X holds for ζ, then any superset of X also holds for ζ. That is, if ζ is violated for an itemset S then it is violated for any subset of S. □

The $sum(S) \geq v(\forall a \in S, a \geq 0, v \geq 0)$ is an example of a *monotone* constraint. Using the same items A, B, and C from above, and by applying the constraint ζ, where ζ is the $sum(S) \geq 500$, then it is enough to find that ABC violates the constraint ζ to stop testing all its sub-patterns as they all violate the same constraint ζ.

Table 2.2 presents commonly used constraints that are either anti-monotone or *monotone*. From the definition of both types of constraints we can conclude that *anti-monotone*

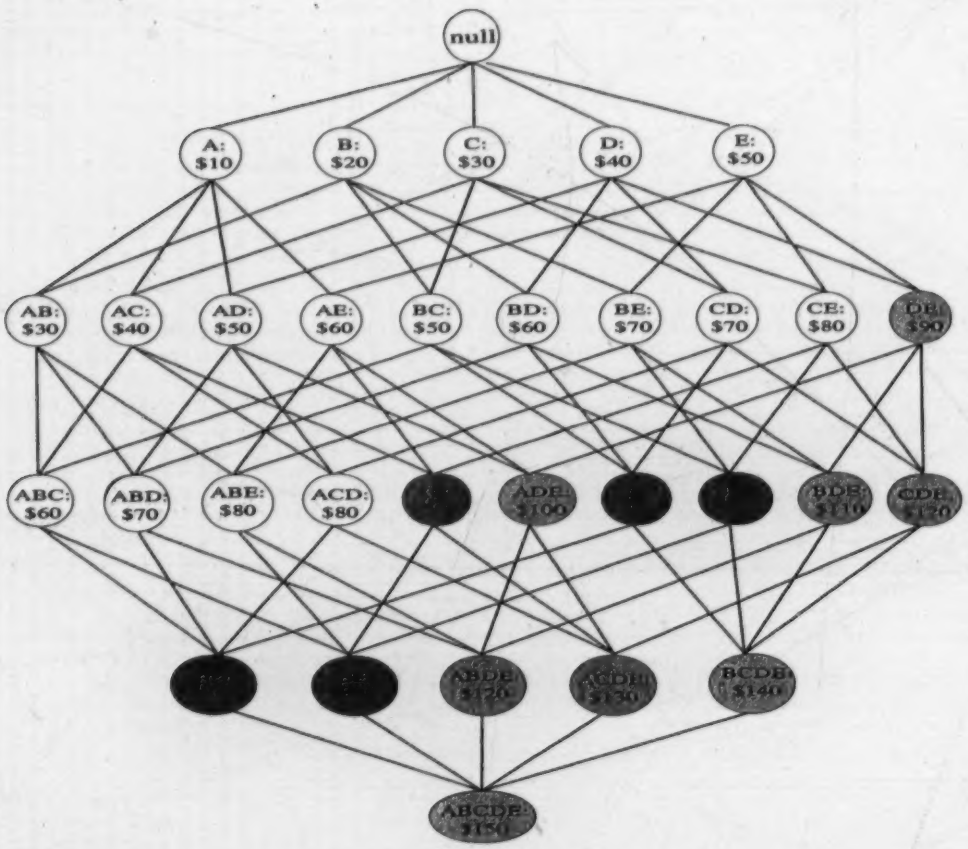


Figure 2.10: Lattice for all possible itemsets from ABCDE with their respective prices.

constraints can be exploited efficiently when the mining-algorithm uses the bottom-up approach, as we can prune any candidate superset if its subset violates the constraint. On the other hand, the *monotone* constraints can be used efficiently when we are using algorithms that follow the top-down approach as we can prune any subset of patterns from the answer set once we find that its superset violates the *monotone* constraint. For example, assume that we have a frequent pattern $ABCDE$, where the prices of items A, B, C, D and E are \$10, \$20, \$30, \$40, and \$50 respectively. Figure 2.10 presents the lattice for all possible frequent patterns that can be generated from $ABCDE$ and with their respective prices. From this figure we can find that we needed to generate and count five patterns of size 1, ten patterns of size 2, ten patterns of size 3, five patterns of size 4, and one pattern of size 5 which makes a total of 31 patterns. If the user wants to find all frequent itemsets that have prices ranging from more than \$50 to less than \$90, there are two alternatives: either using a bottom-up approach and deal with *anti-monotone* constraints and postpone the *monotone* ones, or using a top-down approach and pushing the *monotone* constraints early while

leaving the *anti-monotone* ones as a filter. Let us consider the *anti-monotone* constraint *price of X is less than $90*. We can find that at the second level DE violates the constraint and consequently ADE, BDE, CDE, $ABDE$, $ACDE$, $BCDE$, and $ABCDE$ should not be generated and counted, which saves us from generating seven patterns. At level 3 we find that ACE, BCD and BCE also violate the constraint which means we do not need to generate and count $ABCD$ and $ABCE$. Which means in total by pushing this constraint early we were able to prune directly nine patterns. The second monotone constraint can be applied as a post-mining step to remove any frequent patterns that violate it (i.e. price greater than $50). Figure 2.10 illustrates this pruning. The other alternative is to consider the *monotone* constraints first starting from the long patterns. Once we find for example that AB has a price less than $50, we can directly omit single items A and B. The same applies to AC, AD and BC. After that the anti-monotone constraint can be applied to the generated patterns.

2.5.2 Bi-Directional Pushing of Constraints

Pushing constraints early means considering constraints while mining for patterns rather than postponing the checking of constraints until after the mining process. Given the intrinsic characteristics of existing algorithms for mining frequent itemsets while pushing constraints, either going over the lattice of candidate itemsets top-down or bottom-up, considering all constraints while mining, is difficult. Most algorithms attempt to push either type of constraints during the mining process hoping to reduce the search space in one direction: from subsets to supersets or from supersets to subsets. Dualminer [15] pushes both types of constraints but at the expense of efficiency. Focussing solely on reducing the search space by pruning the lattice of itemsets is not necessarily a winning strategy. While pushing constraints early seems conceptually beneficial, in practice the testing of the constraints can add significant overhead. If the constraints are not selective enough, checking the constraint predicates for each candidate can be onerous. It is thus important that we also reduce the checking frequency. While pruning is to avoid testing candidates that are de facto known to violate constraints, we could also mark and slice parts of the itemset lattice and in this way avoid testing candidates when they are in effect known to satisfy all constraints. In other words, given the definition of constraints, *monotone* or *anti-monotone*, and the *a-priori* property of itemsets, it is possible to effectively determine whether some patterns satisfy or not the given constraints without in reality checking the constraints for each individual pattern. In summary the goal of pushing constraints early is to reduce the itemset search space and eliminate unnecessary processing and data structures while at the same time still limiting the constraint checking.

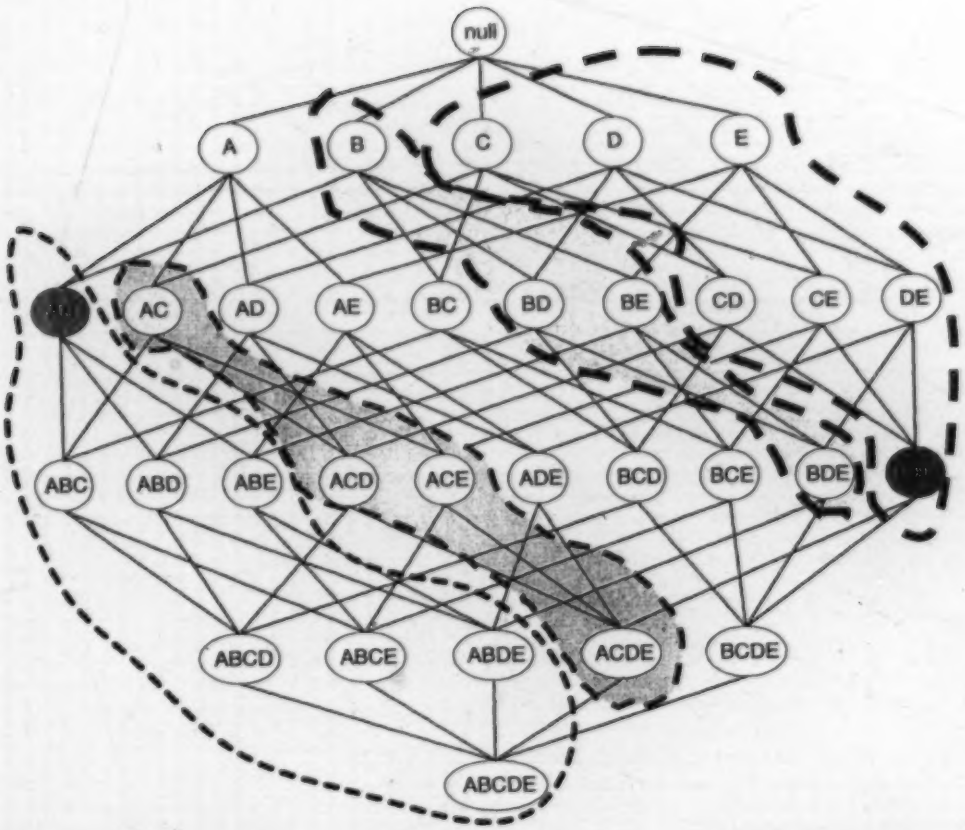


Figure 2.11: Prunning of the lattice: CDE violates *monotone* constraints while AB violates *anti-monotone* constraints. All their subsets and supersets are pruned. BDE & AC satisfy the *monotone* & *anti-monotone* constraints respectively. There is no need to check their subsets and supersets.

2.5.3 State-of-the-Art Algorithms

Mining the frequent patterns with constraints has been studied heavily: The concepts of *monotone* and *anti-monotone* were introduced in [61] to prune the search space. Many algorithms were based on these two concepts:

- Jian Pei et al. [67, 68] have generalized these two classes of constraints and introduced a new succinct class of constraints. In their work they proposed a new algorithm called FIC^M which is an *FP-Growth* based algorithm [48]. This algorithm generates most frequent patterns before pruning them. Its main contribution is that it checks for *monotone* constraints early and once a frequent itemset is found to satisfy the *monotone* constraint, then all itemsets having this item as a prefix are sure to satisfy the constraint and consequently there is no need to apply any checks.

- *Dualminer* [15], is the first algorithm to mine both types of constraints at the same time. However, this algorithm suffers from many practical limitations and performance issues: First, it is built on the top of MAFIA [16] algorithm which produces the set of maximal patterns, and consequently all frequent patterns generated using this model do not have their support attached with each frequent itemset. Second, it assumes that the whole dataset can fit in main memory which is not always the case. Third, their top-down computation exploiting the *monotone* constraint performs in many cases useless tests for a relatively large dataset, which raises many question about the real performance gained in pushing constraint in the *Dualminer* algorithm. In a recent study of parallelizing *Dualminer* by Ting et al. [77], the authors showed that by mining a relatively small sparse dataset made only of 10K transactions consisting of 100K items, the sequential version of *Dualminer* took an inexplicable length of time. Unfortunately the original authors of *Dualminer* did not show any single experiment to depict the execution time of their algorithm but only the reduction in candidate itemsets [15].

- ExAminer [10], a recent strategy dealing with *monotone* and anti-*monotone* constraints, suggests reducing the transactional database input as pre-processing by successively eliminating transactions that violate these constraints and later applying any frequent itemset mining algorithm on the reduced transaction set. The main drawback of this approach is that it is highly I/O bound because of the iterative process needed in re-writing the reduced dataset to disk. This algorithm is also sensitive to the results of the initial *monotone* constraint checking, which is applied to the full transaction. In other words, if the whole transaction satisfies the *monotone* constraint, then no pruning is applied and consequently no gains are achieved while parts of this transaction may not satisfy the same *monotone* constraint. To overcome some of the issues

by Bunchi et al. [10], the same approach has been tested against the *FP-Growth* approach with new effective pruning heuristics [11].

2.6 Parallel Frequent Pattern Mining

Searching for hidden information in the repositories of data is an important goal for decision makers, but finding this knowledge in a "reasonable" time is more significant for them. Speeding up the process of knowledge discovery has become a critical problem, and parallelism is shown to be a potential solution for such problems. But before we go in depth on this parallelism issue, we should note that parallelism is not the only approach that might speed up the data mining process. In fact, other approaches might help in achieving this goal: sampling, attribute selection, restriction of search space, and algorithm or code optimization [40]. Some of these approaches might be used in conjunction with parallelism to achieve the desired speedup.

Efficiency is crucial in knowledge discovery systems, and with the explosive growth of data collection, sequential data mining algorithms have become unacceptable solution to most real size problems. The databases used for knowledge discovery usually contain details of the entire history of a company's standard transactional databases, which presently, for some companies grows past hundreds of gigabytes towards multiples of terabytes formed from hundreds of thousands of different items. This makes it unrealistic for one processor to mine them sequentially, especially when we are dealing with multiple passes over these enormous databases. Dividing the mining tasks among different processors draws a potential solution for the above-mentioned problem especially if this parallelism provides knowledge for decision makers in a reasonable time period and allows them to work effectively.

In this section, we are summarizing most of the existing parallel association mining algorithms, which will be grouped into three main categories based on the candidate sets allocations. The first one accumulates all algorithms that rely on replications of candidate sets. The second one partitions the candidate set. The last one is the set of algorithms that performs a hybrid approach. A summary of the most important exciting parallel association rules mining algorithms follows in this section.

2.6.1 Replication Algorithms

As mentioned above there are three categories of parallel frequent itemset mining algorithms, which are the replications, partitioning and the hybrid approaches. The replication is the simplest approach, where the candidate generation process is replicated and the counting step is performed in parallel where each processor is assigned part of the database to mine. This method suffers mainly from the fact that not all local frequent items are global frequent items, (false positive phenomena).At the same time not all locally infrequent items are

globally infrequent, (false negative phenomena). This method also depends heavily on the memory size. The main algorithms in this class are:

- Count Distribution algorithm [70], can be considered as a parallel formulation of the *Apriori* algorithm. It achieves its parallelism by partitioning the data equally among all processors, where each processor builds an identical hash tree for all candidate keys and mines its own local data to collect the local counts. A global sum reduction (All-To- All communication) occurs at the end of each phase to accumulate the global counts for all candidates, which will be used on the next iteration. This algorithm trades duplication of work with minimal per-iteration communication (an asynchronous broadcast of frequency counts only).

- Parallel Partition algorithm [73], is similar to the Count Distribution in terms of replicating the candidate set among processors. However it differs in the way of counting global counters, where it starts by finding the local counts for each data set. After that, each processor starts to broadcast its local data to accumulate them into the global candidate set. Then the local counts of these candidate set are computed by scanning the local database again. Finally, a communication phase is executed to generate the global counts. This algorithm depends heavily on the size of the local dataset and the skew of the data because it produces a lot of false positives that may dramatically reduce the efficiency. However since it uses a vertical data layout the count phase is considered efficient.

- Fast Distributed Mining algorithm [20], is built over the Count Distribution with an additional optimization steps to reduce the number of candidates considered for counting. The first optimization is achieved due to the fact that Count Distribution uses All-To-All broadcast of the local counts, however the FDM sends the local counts to only one home site per candidate, which consequently reduces the communication cost dramatically. The second optimization of the FDM algorithm involves global pruning that sends the global supports for frequent items with their local support at each partition. These optimization steps for the FDM algorithm introduce some drawbacks such as the need of two rounds of messages in each iteration, one for computing the global support and one for broadcasting the frequent items, which could degrade the performance in the parallel execution.

- Fast Parallel Mining algorithm [20], is based on the FDM algorithm with a goal of eliminating some drawbacks of the FDM algorithm. The main problem of the FDM algorithm where it requires two rounds of messages has been replaced in the FPM algorithm by broadcasting the local support for all processors.

- Parallel Data Mining algorithm [63], is a parallel formulation of the DHP algorithm [62], which is an *Apriori*-like algorithm that differs from the *Apriori* algorithm in its prediction step where it uses hash tables to look ahead into the potential candidates of the next phase. In the PDM algorithm, the hash table is constructed in parallel, where each processor constructs its part of the hash table from its local items. Since each iteration requires the need for the whole hash table, an All-To-All broadcast of the hash table is done to obtain the global count of the item set. This All-To-All simple broadcast for the hash table makes the PDM algorithm very expensive and inefficient. An optimization step is added to overcome this complexity, by simply broadcasting only the potential frequent items. This is done due to the fact that: "An entry in the global hash table will be greater than the support threshold, s, only if at least one processor has its corresponding local entry greater than s/p where p is the number of processors. In Summary PDM is similar to CD algorithm, but efficient parallelizing of hash table construction gives it an edge over the CD algorithm.

2.6.2 Partitioning Algorithms

The second type of parallel algorithms rely on the concept of partitioning the candidate set among processors where each processor handles only a predefined set of candidate items and scans the entire database which might in some data stores reach terabytes of data. In cases of extremely large databases these algorithms pay the cost of the massive I/O scans required by them. In general these algorithms mine relatively small databases with limited memory bandwidth. Some of these algorithms are:

- The Data Distribution [70] algorithm is designed to minimize computational redundancy and maximize use of the memory bandwidth of each node. It works by partitioning the current maximal-frequency itemset candidates (like those generated by *Apriori*) among the nodes. Thus, each node examines a disjointed set of possibilities; however, each node must scan the entire database to examine its candidates. Thus this algorithm trades off a huge amount of communication (to fetch the database partitions stored on other nodes) for better use of machine resources and to avoid duplicated work.

- The Candidate Distribution algorithm [70] is similar to Data Distribution in that it partitions the candidates across nodes, but it attempts to minimize communication by selectively replicating the database among the nodes so that each node can generate global counts from its local databases. This algorithm redistributes the data among processors while scanning the database which reduces its performance compared with the performance of the Count Distribution. The experiments conducted on this algorithm show that it also suffers from bad load balancing. However, the effects of poor

load balancing are mitigated somewhat, since global barriers at the end of each pass are not required.

- Intelligent Data Distribution IDD [46], algorithm was proposed to solve some of the redundant work of the DD algorithm. The first problem that this algorithm addresses is the expensive communication due to the ring-based All-To-All broadcasting for the local portions of the data sets. The IDD algorithm replaces this with only Point-To-Point communication between neighbours, thus eliminating any communication contentions. Second, if the algorithm finds that the candidate set can fit in memory then it switches directly to Count Distribution. Finally the partitioning schema performed in DD algorithm, which produces some redundancy, has been replaced by a single item, prefix based partition, where IDD algorithm before processing the transaction makes sure that it contains the relevant prefix; if not, then it will be discarded.

- Shintani and Kitsuregawa proposed a set of *Apriori*-based parallel algorithms, which is similar to the Count Distributions, Data Distribution, and intelligent distributions. Non Partitioned Apriori [74] are similar to the Count distribution in all its phases except the communication phase, where in the CD algorithm the local counts are exchanged among all processors to produce the global count. However, in the NPA the local count are gathered into one master node. Simple partitioned [74] is almost identical to the Data Distribution algorithm. The Hash Partitioned Apriori and HPA-ELD [74] (Hash Partitioned *Apriori* with Extremely large item sets Duplications) are similar to the IDD algorithm. The new sets of algorithms are introduced with the goal of eliminating the redundant computation in the CD, DD, and IDD algorithms. An example of these improvements is the introduction of a new function to determine the home processor for each frequent candidate and only candidates that are not already on the their home directory will be sent. This means that for all candidate items that are on their home processors an insert function will be applied to insert them into the local hash tree and only the remaining will be sent to their home directories. The HPA-ELD is a variant of HPA algorithm that has been proposed for large databases. The idea of this algorithm emerged from the fact that partitioning the transactions equally does not mean that the candidate sets will be equally distributed among processors. In reality some processors will be holding the most frequent item sets with their high load, leaving other processors that are holding less frequent items sets lightly loaded. HPA-ELD addresses this problem by distributing the highly loaded frequent item sets among all processors to achieve better load balancing among them, and then process these frequent item sets using the Simple partitioned algorithm.

- The Asynchronous Parallel Mining Algorithm [19] is based on DIC algorithm [14]

"Dynamic Itemset Counting" which is a generalization of the *Apriori* algorithm. The DIC algorithm partitions the database into sufficient equal size partitions where each partition should fit in the memory. The DIC strategy relies on the fact that the partitioned data are homogenous (have similar distribution for the frequent itemsets) which consequently causes a decrease in the number of the database scans. However if the data is heterogeneous (the frequent itemsets do not have similar distribution) then this algorithm scans the I/O more than the Apriori. Back to the APM algorithm where this algorithm divides the database into several equal sized partitioned which are usually more than the number of processors. The APM accumulates the local counts of each partition and cluster these counts in K clusters where these clusters are skewed as possible. Finally DIC is applied in parallel for each of the clusters or partitions. During this, a shared prefix tree is built asynchronously among processors to ensure that each partition is homogenous, which is achieved by assigning to each processor an equal mix of virtual partitions from separate clusters, resulting in homogenous processor partitions.

- A new set of algorithms based on lattice traversal were proposed by Zaki [64]. These algorithms require only 3 complete I/O scans of the database in which the first scan generates the item-transaction list which alters the layout of this algorithm into a vertical one. These algorithms try to prune the itemset search space by generating clusters of the related potential maximal frequent items. This cluster pruning is implemented either by equivalence class method or hyper-graph clique method. Each Class of the itemset forms a disjoint sub lattice of the entire itemset lattice, where parallelization is applied in determining the clustering and processes each of these clusters independently by different processors. This algorithm performs dynamic load balancing by estimating the needed time to process each cluster and based on this estimation it makes a decision on how many clusters will be assigned to each processor. One of the main advantages of these set of algorithms is the fact that they only require 3 full database scans; however, they still suffer from drawbacks such as their efficiency relying completely on the number and robustness of the clusters produced and the skewness of the items on the transactions. If few clusters are produced then the algorithm may not be able to fully utilize all the allocated processors, or it may not even be able to apply the dynamic load balancing efficiently, which makes this set of algorithms not scalable for large numbers of processors. It also incurs the expense of replicating part of the databases among processors, and this is highly relevant in cases where the clusters suffer from many overlaps between the frequent items.

- A parallelization of MaxMiner [8] is presented in [21]. The algorithm inherits the

effective pruning of MaxMiner but also its drawbacks. It is efficient for long maximal patterns but not as capable when most patterns are short. It also requires multiple scans of the data making it inefficient for extremely large datasets.

- A PC-cluster based algorithm [69], derived from the sequential *FP-Growth* algorithm [48], exhibits good load balancing. Being a non *Apriori* based approach, the candidacy generation is significantly reduced. However, node-to-node communication is considerable, especially for sending conditional patterns. The algorithm displays good speedup, but it does not scale to extremely large datasets as the larger the dataset, the more conditional patterns are found, and the more node-to-node communication is required.

2.6.3 Hybrid Approach

The previous two categories of algorithms are either pure replications or pure partitioning, where each method has some major issues with its scalability. Most replicated algorithms are sensitive to the size of the transaction set whereas partitioning ones are sensitive to the number of candidates. A new set of algorithms with the purpose of merging the two techniques to diminish the scalability problem as much as possible is called the hybrid approach. These algorithms mainly apply partial replications of the Candidate set. The Hybrid Distribution algorithm [46] combines the ideas of both IDD and CD algorithms. This algorithm partitions the Candidate set into a big enough section and assigns group of processors for each partition. The Count Distribution runs over each one of these partitions and within them the local counts are computed using the IDD algorithm. This algorithm has many advantages over the CD, DD, and IDD algorithms by reducing the database communications into 1/G where G is the number of partitioned groups. It also has a good load balancing by trying to keep each processor busy especially during the later iterations.

2.7 Open-problems and Main Issues

Open-problems still exist with crucial issues such as large data size, high dimensionality, and what we refer to as superfluous search of the lattice and processing. We concede that there are algorithms that perform well in some circumstances. However, to the best of our knowledge, there are no algorithms to date that achieve good results in an environment that combines all or some of the above problems. The open-problems are summed up as follows:

2.7.1 Large Data Size

Devising new scalable algorithms for mining extremely large databases to generate frequent patterns is one of the goals set by the data mining community. To achieve this, speeding up

the existing algorithms gained popularity and relevance. However, the crucial indicator of obtaining scalable algorithms is not necessarily achieved by accelerating the existing ones. The capability of mining extremely large data in a reasonable time is the real factor in determining the algorithm scalability. Larger size problems introduce potential restrictions and ways of measurements in terms of time and space complexity. For time complexity, the growth rate of the algorithm run time, with the increase in problem size, indicates its time complexity and determines its scalability. Space complexity is measured by any new constraints raised due to the increase of the problem size. For example, the absolute main memory size, in which the computation is performed, can cause real limitations.

Current frequent itemset mining algorithms are not scalable for extremely large databases due to many reasons. One of these reasons can be explained by knowing that the Apriori-based algorithms require multiple scans of the databases, even where single or double scans for large databases are considered expensive. New research directions have been proposed based on reducing the number of scans. Such approaches rely on creating memory-based structures for storing frequent items or even the transactions in special data structures mainly prefix trees [50]. In cases of large databases, these structures will certainly not fit in main memory. Consequently, they become a barrier for achieving the goal of mining extremely large databases. A new study by Liv et al [59] showed that each of the existing class of algorithms such as *Apriori*-based, *FP-Growth*, and H-Tree [66] have a dataset size limit that it cannot exceed. For example using [59] hardware the *Apriori* algorithm could not mine datasets with sizes more than 2 million transactions. The *FP-Growth* could not mine datasets made of 4 million transactions using a dimension of ten thousand of items due to the important stack size requirements due to the recursion of the approach. This obviously draws a great deal of attention toward the scalability problem that the current algorithms suffer from.

2.7.2 High Dimensionality

Most current algorithms can only handle few thousands of frequent itemsets, which make them not scalable to mine larger dimension size problems. To illustrate this, the second iteration of the Apriori-based algorithms, which counts the frequency of all pairs of items, has a quadratic complexity. To make it clear, if there are 10^4 frequent singleton items, then the Apriori needs to generate more than 10^7 candidate pairs. Moreover discovering a pattern of size 100 requires generating more than $2^{100} = 10^{30}$ candidates in total [50]. New algorithms have also been proposed to solve this problem in which no frequency generation is needed. These algorithms again require memory structures that make them not scalable.

2.7.3 Superfluous Search of the Lattice

Current algorithms find patterns by using one of the two methods, namely breadth-search or depth-search. Breadth-search can be viewed as a bottom-up approach where the algorithms visit patterns of size $k+1$ after finishing the k sized patterns. The depth-search approach does the opposite where the algorithm starts by visiting patterns of size k before visiting those of size $k-1$. Both methods show some redundancy in mining specific datasets. Weng et al. [78] stated that the breadth-search is not the way to go. They recommend the depth-search method for mining long frequent patterns. It is known that many datasets have short frequent patterns [43]. Even for datasets that have long patterns, it does not mean at all that the patterns generated will be long. Many other frequent patterns are still short, and mining them using depth-search method might result in poor performance. In general, both methods show some efficiency while mining some databases. On the other hand, they showed weaknesses or inefficiency in many other cases. To understand this process fully, we will try to focus on the main advantages and drawbacks of each one of these methods in order to find a way to make use of the best of both of them, and to diminish as much as possible their drawbacks. As an example, in the context for mining for maximal patterns, assume we have a transactional database that is made of a large number of frequent 1-itemsets, and has maximal patterns with relatively small lengths. The trees built from such database are usually deep as they have a large number of frequent 1-itemsets especially if they are made of relatively long transactions. Traversing in depth-search manner would provide us with potential long patterns that end-up non frequent. In such cases, the depth-search method is not favored. However, if the maximal patterns generated are relatively long with respect to the depth of the tree, then the depth-search strategy is favored as most of the potential long patterns that could be found early tend to be frequent. Consequently, many pruning techniques could be applied to reduce the search space. On the other hand, mining transactional databases that reveal long maximal patterns is not favored using breadth-search manner, as such algorithms consume many passes over the database to reach the long patterns. Such algorithms generate many frequent patterns at level k that would be omitted once longer superset patterns at level $k+1$, or $k+l$ for any l, appear. These generation and deletion steps become a bottleneck while mining transactional databases of long frequent patterns using the breadth-search methods.

2.7.4 Observations on Superfluous Processing

Frequent itemset mining algorithms mine the database on a given fixed support threshold. If the support threshold changes, the mining process is repeated. In practice, since the minimum support is not necessarily known and needs tuning, the mining process is interactively repeated with different values for the support threshold. In particular, if the support

is consecutively reduced, k new scans of the database are needed for the *Apriori*-based approaches, and a new memory structure is built for *FP-Growth* like methods. Notice that in each run of these algorithms, previously accumulated knowledge is not taken into account.

Chapter 3

COFI-trees, FP-Tree and the Inverted Matrix

A small cloud may hide both sun and moon.

– Danish Proverb

Existing association-rule-mining algorithms suffer from many problems when mining massive transactional datasets. One major problem is the high memory dependency: either the gigantic data structure built is assumed to fit in main memory, or the recursive mining process is too voracious in memory resources. Another major impediment is the repetitive and interactive nature of any knowledge discovery process. To tune parameters, many runs of the same algorithms are necessary leading to the building of these huge data structures time and again. This chapter proposes two novel ideas:

- a new disk-based association rule mining algorithm called Inverted Matrix [25], which achieves its efficiency by converting its transactional data into a new database layout called Inverted Matrix that prevents multiple scanning of the database during the mining phase, in which finding frequent patterns could be achieved in less than a full scan with random access.
- For each frequent item, a relatively small independent tree called Co-Occurrence Frequent Item Tree, or COFI-tree for short [24], is built summarizing co-occurrences. Then a simple and non recursive mining process mines the COFI-trees.

We show that those COFI-trees can be built either based on Inverted Matrix or on frequent pattern tree data structure [48]. Experimental studies reveal that our COFI-trees and Inverted Matrix approaches outperform *FP-Growth* especially in mining very large transactional databases with a very large number of unique items. Our random access disk-based approach is particularly advantageous in a repetitive and interactive setting.

3.1 Frequent Pattern Tree: Design and Construction

The COFI-tree approach we propose consists of two main stages. Stage one is the construction of the Frequent Pattern tree or the Inverted Matrix (discussed later in this chapter) and stage two is the actual mining for these data structures.

3.1.1 Construction of the Frequent Pattern Tree

The goal of this stage is to build the compact data structures called Frequent Pattern Tree [48]. This construction is done in two phases, where each phase requires a full I/O scan of the dataset. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

This phase starts by enumerating the items appearing in the transactions. After enumerating these items (i.e. after reading the whole dataset), infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. Phase 2 would construct a frequent pattern tree.

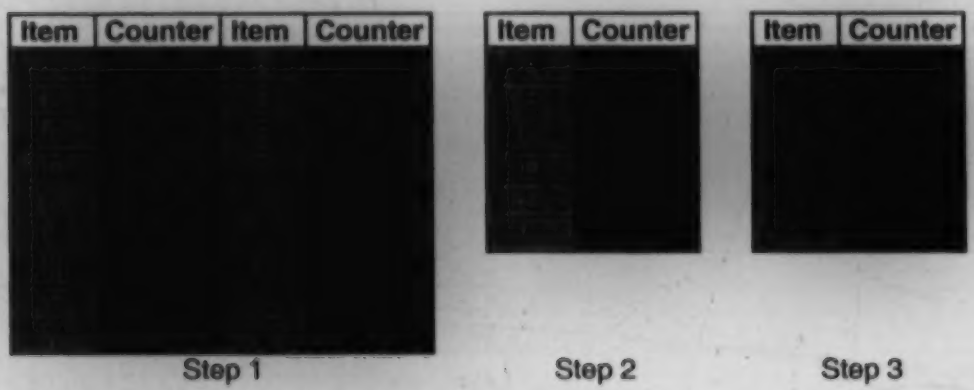


Figure 3.1: Steps of phase 1.

Phase 2 of constructing the Frequent Pattern tree structure is the actual building of this compact tree. This phase requires a second complete I/O scan from the dataset. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Tree as follows: for the first item on the sorted transactional dataset, check if it exists as one of the children of the root. If it exists then increment the support for this node. Otherwise, add a new node for this item as a child for the root node with 1 as support. Then, consider the current item node as the new temporary root and

repeat the same procedure with the next item on the sorted transaction. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

For illustration, we use an example with the transactions shown in Figure 2.3. Set the minimum support threshold to 4. Phase 1 starts by accumulating the support for all items that occur in the transactions. Step 2 of phase 1 removes all non frequent items, in our example ($G, H, I, J, K, L, M, N, O, P, Q$ and R), leaving only the frequent items (A, B, C, D, E, and F). Finally all frequent items are sorted according to their support to generate the sorted frequent 1-itemset. This last step ends phase 1 of the COFI-tree algorithm and starts the second phase. In phase 2, the first transaction (A, G, D, C, B) read is filtered to consider only the frequent items that occur in the header table (i.e. A, D, C and B). This frequent list is sorted according to the items' supports (A, B, C and D). This ordered transaction generates the first path of the FP-Tree with all item-node support initially equal to 1. A link is established between each item-node in the tree and its corresponding item entry in the header table. The same procedure is executed for the second transaction (B, C, H, E, and D), which yields a sorted frequent item list (B, C, D, E) that forms the second path of the FP-Tree. Transaction 3 (B, D, E, A, and M) yields the sorted frequent item list (A, B, D, E) that shares the same prefix (A, B) with an existing path on the tree. Item-nodes (A and B) support is incremented by 1 making the support of (A) and (B) equal to 2 and a new sub-path is created with the remaining items on the list (D, E) all with support equal to 1. The same process occurs for all transactions until we build the FP-Tree for the transactions given in Figure 2.3. Figure 3.2 shows the result of the tree building process.

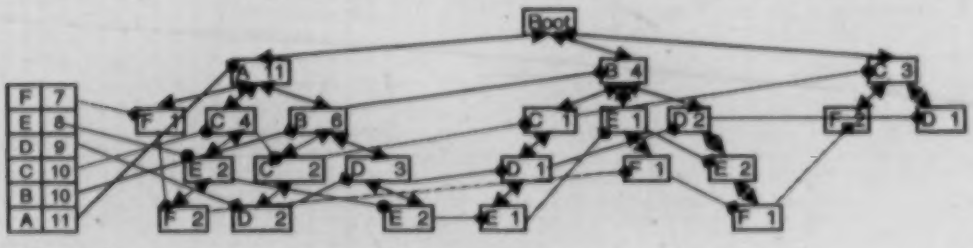


Figure 3.2: Frequent Pattern Tree

3.2 Co-Occurrence Frequent-Item-trees: Construction, Pruning and Mining

Our approach for computing frequencies relies first on building independent relatively small trees for each frequent item in the header table of the FP-Tree called COFI-trees. A pruning technique is applied to remove all non frequent items with respect to the main frequent item

of the tested COFI-tree. Then we mine separately each one of the trees as soon as they are built, minimizing the candidacy generation and without building conditional sub-trees recursively. The trees are discarded as soon as they are mined. At any given time, only one COFI-tree is present in main memory.

3.2.1 Construction of the Co-Occurrence Frequent-Item-trees

The small COFI-trees we build are similar to the conditional FP-Trees in general in the sense that they have a header with ordered frequent items and horizontal pointers pointing to a succession of nodes containing the same frequent item, and the prefix tree per se with paths representing sub-transactions. However, the COFI-trees have bidirectional links in the tree allowing bottom-up scanning as well, and the nodes contain not only the item label and a frequency counter, but also a participation counter as explained later in this section. The COFI-tree for a given frequent item x contains only nodes labeled with items that are more frequent or as frequent as x.

To illustrate the idea of the COFI-trees, we will explain step by step the process of creating COFI-trees for the FP-Tree of Figure 3.2. With our previous example, the first Co-Occurrence Frequent Item tree is built for item F as it is the least frequent item in the header table. In this tree for F-COFI-tree, all frequent items which are more frequent than F and share transactions with F participate in building the tree. This can be found by following the chain of item F in the FP-Tree structure. The F-COFI-tree starts with the root node containing the item in question, F. For each sub-transaction or branch in the FP-Tree containing item F with other frequent items that are more frequent than F which are parent nodes of F, a branch is formed starting from the root node F. The support of this branch is equal to the support of the F node in its corresponding branch in FP-Tree. If multiple frequent items share the same prefix, they are merged into one branch and a counter for each node of the tree is adjusted accordingly. Figure 3.3 illustrates all COFI-trees for frequent items of Figure 3.2.

In Figure 3.3, the rectangle nodes are nodes from the tree with an item label and two counters. The first counter is a *support-count* for that node while the second counter, called *participation-count*, is initialized to 0 and is used by the mining algorithm discussed later, a horizontal link which points to the next node that has the same *item-name* in the tree, and a bi-directional vertical link that links a child node with its parent and a parent with its child. The bi-directional pointers facilitate the mining process by making the traversal of the tree easier. The squares are actually cells from the header table as with the FP-Tree. This is a list made of all frequent items that participate in building the tree structure sorted in ascending order of their global support. Each entry in this list contains the *item-name*, *item-counter*, and a *pointer* to the first node in the tree that has the same *item-name*.

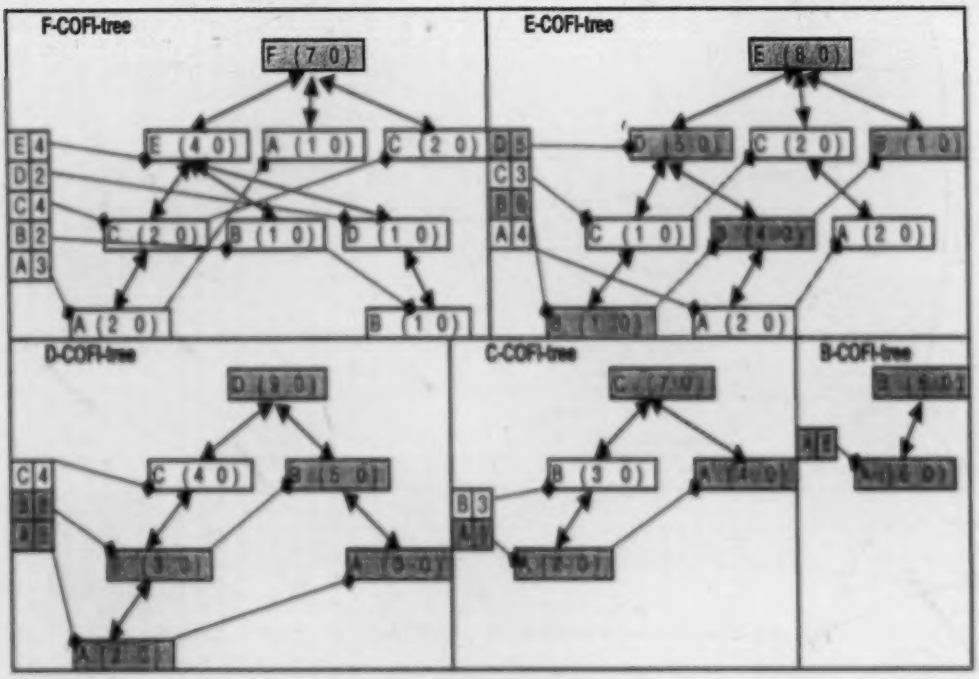


Figure 3.3: COFI-trees

To explain the COFI-tree building process, we will highlight the building steps for the F-COFI-tree in Figure 3.3. Frequent item F is read from the header table and its first location in the FP-Tree is located using the pointer in the header table. The first location of item F indicates that it shares a branch with item A, with support = 1 for this branch as the support of the F-item is considered the support for this branch (following the upper links for this item). Two nodes are created, for FA: 1. The second location of F indicate a new branch of FECA:2 as the support of F=2. Three nodes are created for items ECA with support = 2. The support of the F node is incremented by 2. The third location indicates the sub-transaction FEB:1. Nodes for F and E already exist and only a new node for B is created as a another child for E. The support for all these nodes are incremented by 1. B becomes 1, E becomes 3 and F becomes 4. FEDB:1 is read after that, FE branch already exists and a new child branch for DB is created as a child for E with support = 1. The support for E nodes becomes 4, F becomes 5. Finally FC:2 is read, and a new node for item C is created with support =2, and F support becomes 7. Like with FP-Trees, the header constitutes a list of all frequent items to maintain the location of first entry for each item in the COFI-tree. A link is also made for each node in the tree that points to the next location of the same item in the tree if it exists. The mining process starts by applying a pruning described in the following section. This pruning, based on an *anti-monotone*

property, reduces the candidacy generation step of the mining process. This mining process is the last step done on the F-COFI-tree before removing it and creating the next COFI-tree for the next item in the header table.

3.2.2 Pruning the COFI-trees

In this section we are introducing a new *anti-monotone* property called global frequent/local non frequent property. This property is similar to the *apriori* one in the sense that it eliminates at the i^{th} level all non frequent items that will not participate in the $(i+1)$ level of candidate itemsets generation. The difference between the two properties is that we extended our property to eliminate also frequent items which are among the i-itemset and we are sure that they will not participate in the $(i+1)$ candidate set. The *apriori* property states that *all nonempty subsets of a frequent itemset must also be frequent*. An example is given later in this section to illustrate both properties.

In our approach, we are trying to find all frequent patterns with respect to one frequent item, which is the base item of the tested COFI-tree. We already know that all items that participate in the creation of the COFI-tree are frequent with respect to the global transaction database, but that does not mean that they are also locally frequent with respect to the based item in the COFI-tree. The global frequent/local non frequent property states that *all nonempty subsets of a frequent itemset with respect to item A, must also be frequent with respect to item A*.

In our example we can find that all items that participate in the creation of the F-COFI-tree are locally not frequent with respect to item F as the support for all these items are not greater than the support threshold σ which is equal to 4. From knowing this, there will be no need to mine the F-COFI-tree, as we already know that no frequent patterns other than the item F will be generated. We can extend our knowledge at this stage to note that item F will not appear in any of the frequent patterns. The COFI-tree for item E indicates that only items D, and B are frequent with respect to item E, which means that there will be no need to test patterns as EC, and EA. The COFI-tree for item D indicates that item C will be eliminated as it is not frequent with respect to item D. C-COFI-tree ignores item B for the same reason.

To sum up the *apriori* property states in our example of 6 1-frequent itemsets that we need to generate 15 2-Candidate itemsets which are (A,B), (A,C), (A,D), (A,E), (A,F), (B,C), (B,D), (B,E), (B,F), (C,D), (C,E), (C,F), (D,E), (D,F), (E,F), using our property we have eliminated 9 patterns which are (A,E), (A,F), (B,C), (B,F), (C,D), (C,E), (C,F), (D,F), (E,F) leaving only 6 patterns to test which are (A,B), (A,C), (A,D), (B,D), (B,E), (D,E). This pruning technique is reflected during the mining process as nodes for non locally frequent items are discarded directly. In Figure 3.3, all locally frequent nodes with respect

to COFI-trees are shaded. Figure 3.4 illustrates the pruned COFI-trees of Figure 3.3.

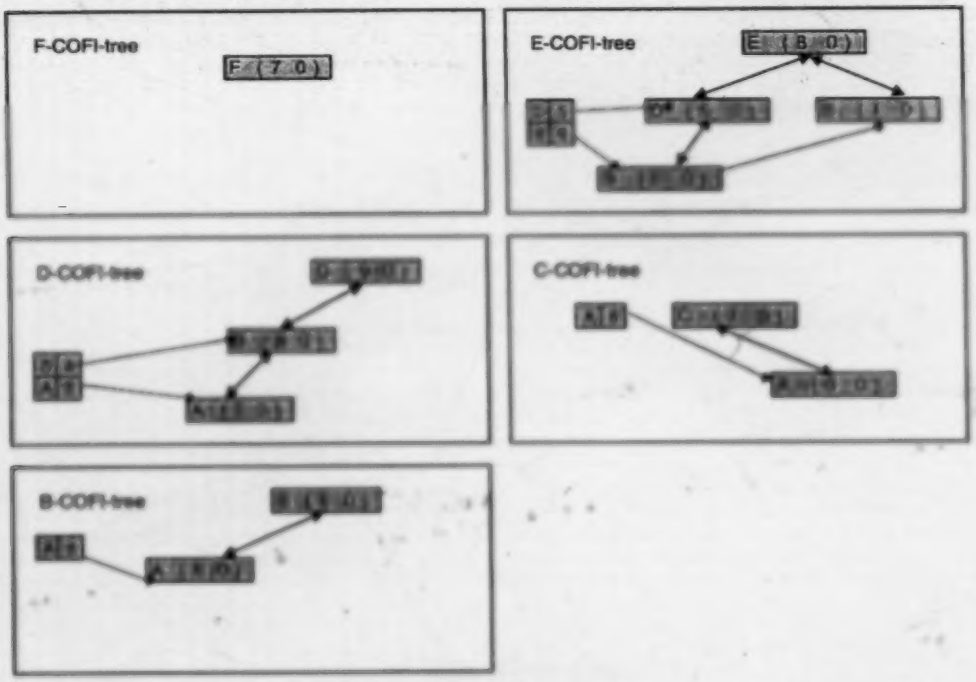


Figure 3.4: Pruned COFI-trees

3.2.3 Mining the COFI-trees

The COFI-trees of all frequent items are not constructed together. Each tree is built, mined, then discarded before the next COFI-tree is built. The mining process is done for each tree independently with the purpose of finding all frequent k-itemset patterns in which the item on the root of the tree participates.

Steps to produce frequent patterns related to the E item for example, as the F-COFI-tree, will not be mined based on the pruning results we found on the previous step, are illustrated in Figure 3.5 assume using the non pruned COFI-trees. From each branch of the tree, using the *support-count* and the *participation-count*, candidate frequent patterns are identified and stored temporarily in a list. The non frequent ones are discarded at the end when all branches are processed. The mining process for the E-COFI-tree starts from the most locally frequent item in the header table of the tree, which is item B, as item A is pruned. Item B exists in three branches in the E-COFI-tree which are (B:1, C:1, D:5 and E:8), (B:4, D:5, and E:8) and (B:1, and E:8). The frequency of each branch is the frequency of the first item in the branch minus the participation value of the same node. Item B in the first branch has a frequency value of 1 and participation value of 0 which

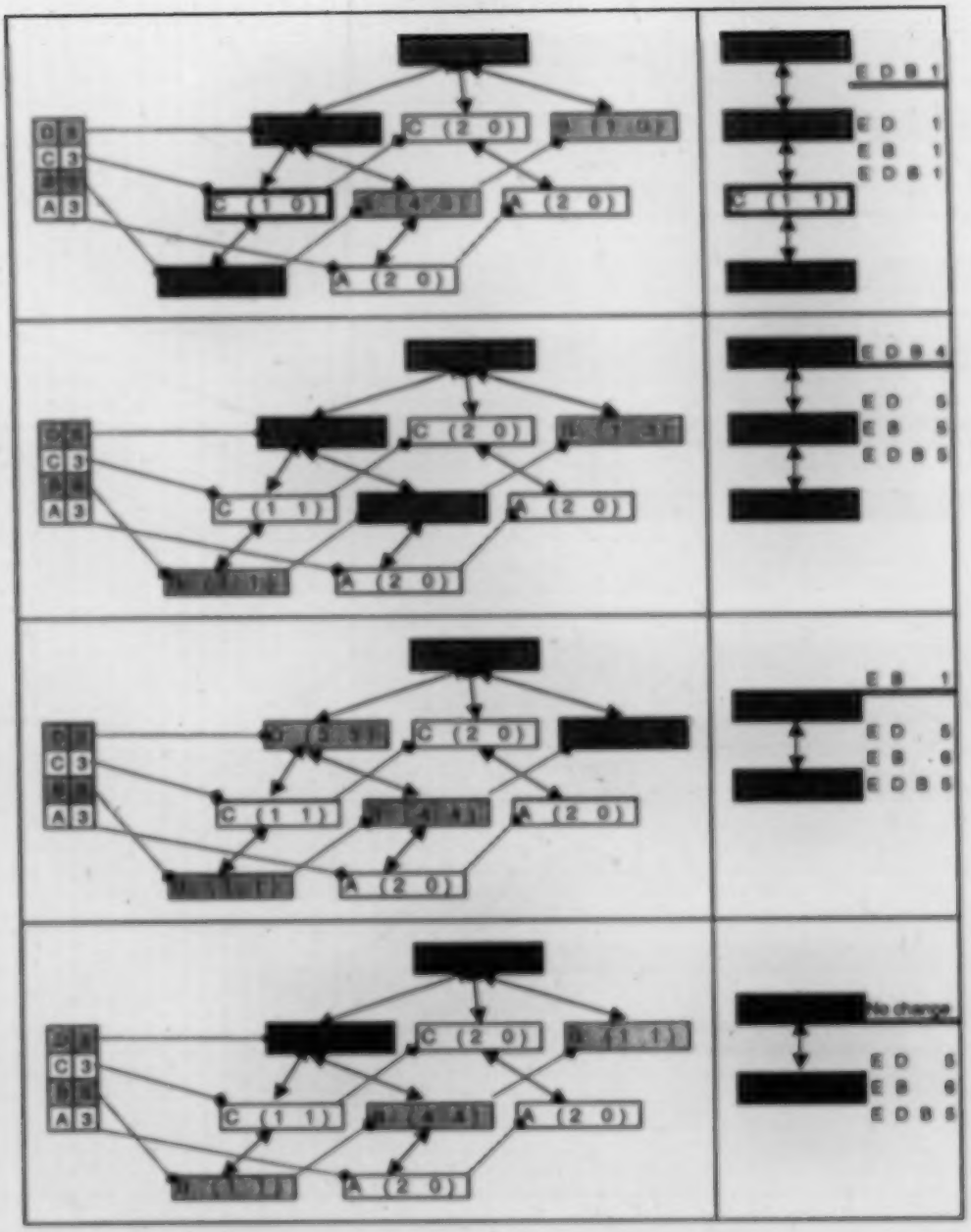


Figure 3.5: Steps needed to generate frequent patterns related to item E

makes the first pattern EDB frequency equal to 1. The participation values for all nodes in this branch are incremented by 1, which is the frequency of this pattern. In the first pattern EDB: 1, we need to generate all sub-patterns that item E participates in which are ED:1 EB:1 and EDB:1. The second branch that has B generates the pattern EDB: 4 as the frequency of B on this branch is 4 and its participation value is equal to 0. All participation values on these nodes are incremented by 4. Sub-patterns are also generated from the EDB pattern which are ED: 4 , EB: 4, and EDB: 4. All patterns already exist with support value equals to 1, and only updating their support value is needed to make it equal to 5. The last branch EB:1 will generate only one pattern which is EB:1, and consequently its value will be updated to become 6. The second frequent item in this tree, "D" exists in one branch (D: 5 and E: 8) with participation value of 5 for the D node. Since the participation value for this node equals to its support value, then no patterns can be generated from this node. Finally all non frequent patterns are omitted leaving us with only frequent patterns that item E participates in which are ED:5, EB:6 and EBD:5. The COFI-tree of Item E can be removed at this time and another tree can be generated and tested to produce all the frequent patterns related to the root node. The same process is executed to generate the frequent patterns. The D-COFI-tree is created after the E-COFI-tree. Mining this tree generates the following frequent patterns: DB:8, DA:5, and DBA:5. C-COFI-tree generates one frequent pattern which is CA:6. Finally, the B-COFI-tree is created and the frequent pattern BA:6 is generated. Algorithm 1 illustrates the creation and mining steps of the COFI-trees.

3.3 COFI-tree: Experimental Evaluations

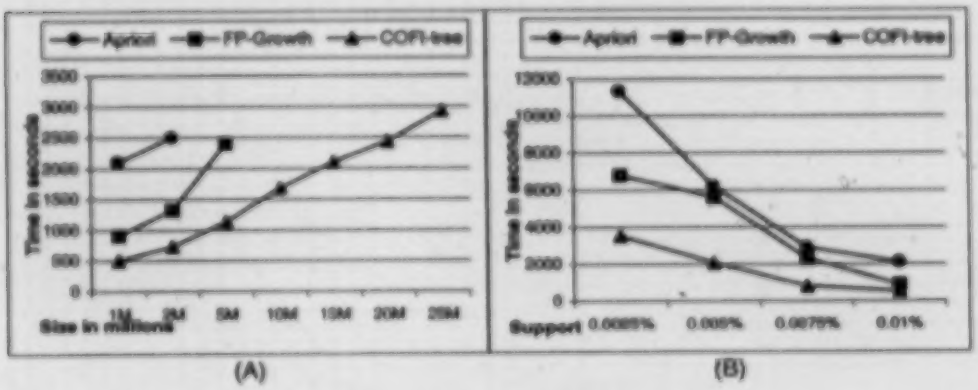


Figure 3.6: Computational performance and scalability

To test the efficiency of the COFI-tree approach, we conducted experiments comparing our approach with two well-known algorithms namely: *Apriori* and *FP-Growth*. To avoid

Algorithm 1 Creating and Mining COFI-trees

Input: FP-Tree and a minimum support threshold σ.
Output: Full set of frequent patterns

A = the least frequent item on the header table of FP-Tree

while There are still frequent items **do**
  Create a root node for the (A)-COFI-tree with both *frequency.count* and *participation.count* = 0
  C is the path from item A to the root
  while C exists **do**
    Items on C form a prefix of the (A)-COFI-tree
    if the prefix is new **then**
      frequency.count= frequency of (A) node
      participation.count = 0 for all nodes in the path
    else
      Adjust the *frequency.count* of the already exist part of the path
    end if
    find the next node for item A in the FP-Tree
  end while
  MineCOFI-tree (A)
  Release (A) COFI-tree
  A = next frequent item from the header table
end while

Algorithm 2 MineCOFI-tree

nodeA = select_next_node {Selection of nodes starts with the node of most locally frequent item and following its chain, then the next less frequent item with its chain, until we reach the least frequent item in the *Header list* of the (A)-COFI-tree}
while there are still nodes **do**
  D = set of locally frequent nodes from nodeA to the root
  F = nodeA.*frequency.count*−nodeA.*participation.count*
  Generate all Candidate patterns X from items in D. Patterns that do not have A will be discarded
  Patterns in X that do not exist in the A-Candidate List will be added to it with frequency = F otherwise just increment their frequency with F
  Increment the value of *participation.count* by F for all items in D
  nodeA = select_next_node
end while
Based on support threshold σ remove non frequent patterns from A Candidate List.

implementation bias, third party *Apriori* implementation, by Christian Borgelt [12], and *FP-Growth* [48] written by its original authors are used. The experiments were run on a 733-Mhz machine with a relatively small RAM of 256MB.

Transactions were generated using an IBM synthetic data generator [51]. We conducted different experiments to test the COFI-tree algorithm when mining extremely large transactional databases. We tested the applicability and scalability of the COFI-tree algorithm. In one of these experiments, we mined using a support threshold of 0.01% transactional databases of sizes ranging from 1 million to 25 million transactions with an average transaction length of 24 items. The dimensionality of the 1 and 2 million transaction dataset was 10,000 items while the datasets ranging from 5 million to 25 million transactions had a dimensionality of 100,000 unique items. Figure 3.6A illustrates the comparative results obtained with *Apriori*, *FP-Growth* and the COFI-tree. *Apriori* failed to mine the 5 million transactional database and FP-Growth could not mine beyond the 5 million transaction mark. The COFI-tree, however, demonstrates good scalability as this algorithm mines 25 million transactions in 2921s (about 48 minutes). None of the tested algorithms, or reported results in the literature reaches such a big size.

To test the behaviour of the COFI-tree vis-à-vis different support thresholds, a set of experiments was conducted on a database size of one million transactions, with 10,000 items and an average transaction length of 24 items. The mining process tested different support levels, which are 0.0025% that revealed almost 125K frequent patterns, 0.005% that revealed nearly 70K frequent patterns, 0.0075% that generated 32K frequent patterns and 0.01% that returned 17K frequent patterns. Figure 3.6B depicts the time needed in seconds for each one of these runs. The results show that the COFI-tree algorithm outperforms both *Apriori* and *FP-Growth* algorithms in all cases.

3.4 Inverted Matrix Layout

As discussed in the previous chapter the transaction layout is the method in which items in transactions are formatted in the database. Currently, there are two approaches: the horizontal approach and the vertical approach. In this section a new transactional layout called Inverted Matrix is presented and compared with the existing two methods horizontal and vertical.

Frequent itemset mining algorithms mine the database based on a given fixed support threshold. If the support threshold changes, the mining process is repeated. In practice, since the minimum support is not necessarily known and needs tuning, the mining process is interactively repeated with different values for the support threshold. In particular, if the support is consecutively reduced, k new scans of the database are needed for the *Apriori*-based approaches, and a new memory structure is built for *FP-Growth* like methods. Notice

that in each run of these algorithms, previous accumulated knowledge is not taken into account. For instance, in the simple transactional database of Figure 3.7.A, where each row represents a transaction (called horizontal layout), we can observe that when changing support one can avoid reading some entries. If the support level is greater than 4, then Figure 3.7B highlights all frequent items that need to be scanned and computed. Non circled items in Figure 3.7.B are not included in the generation of the frequent items, and reading them becomes useless. It is known that all of the existing algorithms scan the whole database, frequent and non frequent items more than once generating a huge amount of useless work [59, 47, 49]. We call this superfluous processing. Figure 3.7.C represents what we actually need to read and compute from the transactional database based on a support greater than 4. Obviously, this may not be possible with this horizontal layout, but with a vertical layout avoiding these useless reads is possible.

T#	Items				
T1	A	B	C	D	E
T2	A	E	C	H	G
T3	B	C	D	A	E
T4	F	A	H	G	J
T5	A	B	C	E	I
T6	K	A	E	I	C
T7	A	H	E	G	I
T8	K	L	M	N	O
T9	L	R	Q	A	O
T10	P	N	B	A	M

(A)

T#	Items				
T1	**A**	B	**C**	D	**E**
T2	**A**	**E**	**C**	H	G
T3	B	**C**	D	**A**	**E**
T4	F	**A**	H	G	J
T5	**A**	B	**C**	**E**	I
T6	K	**A**	**E**	I	**C**
T7	**A**	H	**E**	G	I
T8	K	L	M	N	O
T9	L	R	Q	**A**	O
T10	P	N	B	**A**	M

(B)

T#	Items		
T1	A	C	E
T2	A	E	C
T3	C	A	E
T4	A		
T5	A	C	E
T6	A	E	C
T7	A	E	
T9	A		
T10	A		

(C)

Figure 3.7: A: Transactional database. (B): Frequent items circled. (C): Needed Items to be scanned, $\sigma > 4$.

The Inverted Matrix layout combines the two previously mentioned layouts with the purpose of making use of the best of the two approaches and reducing their drawbacks as much as possible. The idea of this approach is to associate each item with all transactions in which it occurs (i.e. an inverted index), and to associate each transaction with all its items using pointers. Similar to the vertical layout, the item is the key of each record in this layout. The difference between this layout and the vertical layout seen previously is that each attribute on the Inverted Matrix is not the transaction ID, but a pointer that points to the location of the next item on the same transaction. The transaction ID could

Table 3.1: Phase 1, Frequency of each item

Item	Frequency	Item	Frequency	Item	Frequency
P	1	F	1	Q	1
R	1	J	1	O	2
D	2	K	2	L	2
M	2	N	2	I	3
G	3	H	3	B	4
C	5	E	6	A	9

be preserved in our layout, but since it is not needed for the purpose of frequent itemset mining, it is discarded. The pointer is a pair where the first element indicates the address of a row in the matrix and the second element indicates the address of a column. Each row in the matrix has an address (sequential number in our illustrative example) and is prefixed by the item it represents with its frequency in the database. The lines are ordered in ascending order of the frequency of the item they represent. Table 3.2 represents the Inverted Matrix corresponding to the transactional database from Figure 3.7.A. To mine this Inverted Matrix with $\sigma > 4$ only sub transactions in Figure 3.7.C need to be scanned. Consequently, part of the Inverted Matrix is scanned which is presented in Table 3.3.

Table 3.2: Inverted Matrix

loc	Index	Transactional Array								
		1	2	3	4	5	6	7	8	9
1	(P,1)	(10,2)								
2	(F,1)	(5,1)								
3	(Q,1)	(4,1)								
4	(R,1)	(6,2)								
5	(J,1)	(13,2)								
6	(O,2)	(8,2)	(9,2)							
7	(D,2)	(15,1)	(15,2)							
8	(K,2)	(12,2)	(9,1)							
9	(L,2)	(10,1)	(18,7)							
10	(M,2)	(11,1)	(11,2)							
11	(N,2)	(ϕ,ϕ)	(15,4)							
12	(I,3)	(15,3)	(16,5)	(13,3)						
13	(G,3)	(14,1)	(14,2)	(14,3)						
14	(H,3)	(16,2)	(17,4)	(17,6)						
15	(B,4)	(16,1)	(16,3)	(16,4)	(18,9)					
16	(C,5)	(17,1)	(17,2)	(17,3)	(17,4)	(17,5)				
17	(E,6)	(18,1)	(18,2)	(18,3)	(18,5)	(18,6)	(18,7)			
18	(A,9)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)

Building this Inverted Matrix is done in two phases: Phase one scans the database once to find the frequency of each item and orders them into ascending order, such as in Table 3.1 for our illustrative example. The second phase scans the database again once to sort each transaction into ascending order according to the frequency of each item, and then fills in

Table 3.3: scanned Inverted Matrix with $\sigma > 4$

loc	Index	Transactional Array								
		1	2	3	4	5	6	7	8	9
16	(C,5)	(17,1)	(17,2)	(17,3)	(17,4)	(17,5)				
17	(E,6)	(18,1)	(18,2)	(18,3)	(18,5)	(18,6)	(18,7)			
18	(A,9)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)	(ϕ,ϕ)

the matrix appropriately. To illustrate the process, let us consider the construction of the matrix in Table 3.2. The first column *loc* is a simple sequential count identifying location, i.e the row in the matrix. The column *index* contains the 1-itemset with their frequency ordered in descending order by their frequency. The first transaction in Figure 3.7A has items (A, B, C, D, E). This transaction is sorted into (D, B, C, E, A) based on the item frequencies in Table 3.1 built in the first phase of the process. Item D has the physical location row 7 in the Inverted Matrix in Table 3.2, B has the location row 15, the location of C is row 16, E is in row 17 and finally A is in row 18. This is according to the vertical approach. Item D has a link to the first empty slot in the transactional array of item B that is 1. Consequently, (15,1) entry is added in the first slot of item D to point to the first empty location in the transactional array of B. At the First empty location of B (15,1) an entry is added to point to the first empty location of the next item C that is (16,1). The same process occurs for all items in the transaction. The last item of the transaction, item A produces an entry with pointer null (ϕ,ϕ). The same is performed for every transaction.

Building the Inverted Matrix is assumed to be pre-processing of the transactional database. For a given transactional database, it is built once and for all. The next section presents an algorithm for mining association rules (or frequent itemsets) directly from this matrix. The basic idea is straight forward. For example, if the user decides to find all frequent patterns with support greater than 4, it suffices to start the mining process from location row 16. Row 16 represents the item C which has the frequency 5. Since the lines of the matrix are ordered, along with C, only the items that appear after C are frequent. All the other items are irrelevant for this particular support threshold. By following in the Inverted Matrix the chain of items starting from the C location, we can rebuild parts of the transactions that contain only the frequent items. Thus, we avoid the superfluous processing mentioned before. Figure 3.8 represents the sub-transactions that can be generated from the Inverted Matrix of Table 3.2 by following the chains starting from location at row 16. The mining algorithm described in the next section targets these sub-transactions, and passes over all other parts dealing with de-facto non frequent items. The sub-transactions of frequent items such as in Figure 3.8 are never built at once. As will be explained in the next section, these sub-transactions are considered one frequent item at a time. In other words, using the Inverted Matrix, for each frequent item x, the algorithm would identify the sub-transactions of

frequent items that contain x. These sub-transactions are then represented in a COFI-tree, which is mined individually.

C	E	A
C	E	A
C	E	A
C	E	A
C	E	A
E	A	

(A)

Frequent Items	Occurs
C E A	5
E A	1

(B)

Figure 3.8: Sub-transactions with items having $\sigma > 4$. (A) List of sub-transactions; (B) Condensed list.

3.5 COFI-trees Inverted Matrix Based: Design and Construction

Our approach for generating frequent patterns here relies first on reading sub-transactions for frequent items directly from the Inverted Matrix, then building independent relatively small trees for each frequent item in the transactional database. We mine separately each one of the trees as soon as they are built, with minimizing the candidacy generation and without building conditional sub-trees recursively. The trees are discarded as soon as they are mined. These small trees we build, COFI-trees, are discussed earlier in this chapter.

Table 3.4: Example of Sub-transactions with frequent items

Frequent items	Occurs together
CD	2
CB	1
EA	2
FB	2
CDA	1
CBA	4

To illustrate the idea of the COFI-trees based on Inverted Index, let us consider an example of sub-transactions of frequent items. Assume we have a transactional database that has the following frequent items (A, B, C, D, E, and F), where A is the most frequent item, and F is the least frequent item in the database. Assume also that these frequent items occur in the database following the scenario of Table 3.4. These sub-transactions are generated from a given Inverted Matrix. To generate the frequent 2-itemsets, the *Apriori* algorithm would need to generate 15 different patterns out of the 6 items $\{A, B, C, D, E, F\}$.

Finding the frequency of each pattern and removing the non frequent ones is necessary before even considering the candidate 3-itemsets. In our approach, itemsets of different sizes are found simultaneously. In particular, for each given frequent 1-itemset we find all frequent k-itemsets that subsume it. For this, a COFI-tree is built for each frequent item except the most frequent one, starting from the least frequent. No tree is built for the most frequent item since by definition a COFI-tree of an item x contains items that are more frequent than x.

With our example, the first COFI-tree is built for item F. In this tree for F, all frequent items which are more frequent than F and share transactions with F participate in building the tree. The tree starts with the root node containing the item in question, F. For each sub-transaction containing item F with other frequent items that are more frequent than F, a branch is formed starting from the root node F. If multiple frequent items share the same prefix, they are merged into one branch and an account for each node of the tree is adjusted accordingly. Figure 3.9 illustrates all COFI-trees for frequent items of Table 3.4. In Figure 3.9, the round nodes are nodes from the tree with an item label and two counters.

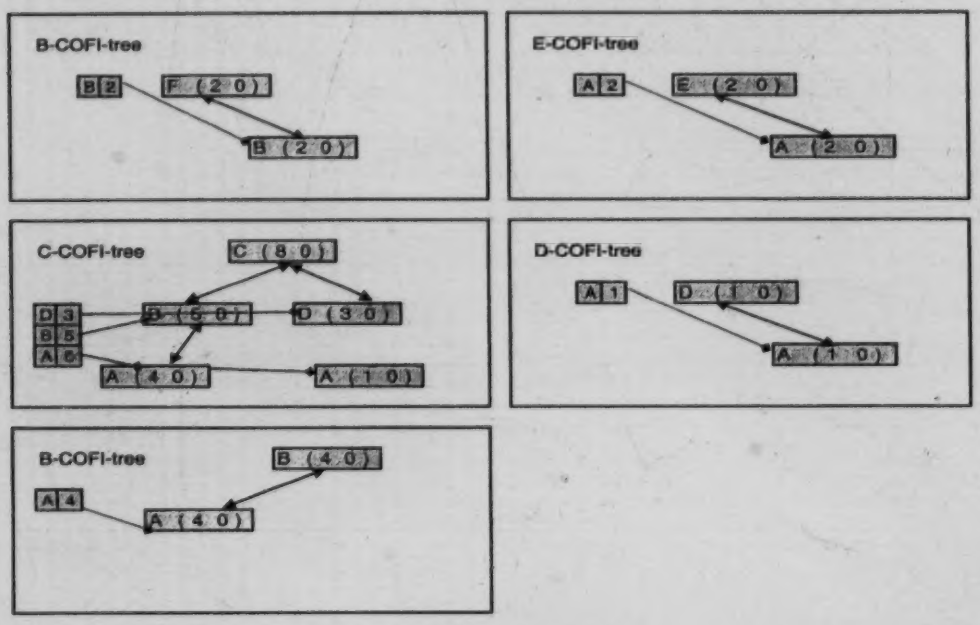


Figure 3.9: COFI-trees

The first counter is a support for that node while the second counter, called participation-count, is initialized to 0 and is used by the mining algorithm discussed later. The nodes also have pointers: a horizontal link which points to the next node that has the same *item-name* in the tree, and a bi-directional vertical link that links a child node with its parent and a parent with its child. The bi-directional pointers facilitate the mining process by making

the traversal of the tree easier. The squares are actually cells from the header table as with the FP-Tree. This is a list made of all frequent items that participate in building the tree structure sorted in ascending order of their global support. Each entry in this list contains the *item-name* and a pointer to the first node in the tree that has the same item-name.

Notice that the COFI-tree for F, Figure 3.9, is made of only two nodes: the root node containing F and one child node for B with frequency = 2, this is because item F occurs twice only with item B in the database presented in Table 3.4. The same thing happens with item E, but it occurs with item A twice. Item C occurs with 3 items, namely A, B and D, and consequently 4 nodes are created as CBA: 4 forms one branch with support = 4 for each node in the branch. CDA: 1 creates another branch with support =1 for the branch except node C as its support becomes 5 (4+1). Pattern CD: 2 already has a branch built, so only the frequency is updated, C becomes 7, and D becomes 3. Finally CB: 1 already shares the same prefix with an existing branch so only counters are updated and thus C becomes 8 and B becomes 5. The D tree is made of one branch as item D occurs once with an item that is more frequent than D, which is in DA: 1 in CDA: 1. Finally item B occurs 4 times with item A from CBA: 4 (C is ignored in the last two cases as it is less frequent than B and A). The header in each tree, like with FP-Trees, constitutes a list of all frequent items to maintain the location of first entry for each item in the COFI-tree. A link is also made for each node in the tree that points to the next location of the same item in the tree if it exists.

The COFI-trees of all frequent items are not constructed together. Each tree is built, mined, then discarded before the next COFI-tree is built. The mining process is done for each tree independently with the purpose of finding all frequent k-itemset patterns that the item on the root of the tree participates in. A Top-Down approach is used to generate and compute maximum n patterns at a time, where n is the number of nodes in the COFI-tree that is being mined excluding the root node of the tree. The frequency of other sub-patterns can be deduced from their parent patterns without counting their occurrences in the database.

Steps needed to produce frequent patterns related to each COFI-tree are similar to the ones described earlier in Section 3.2.3.

3.6 Inverted Matrix Algorithm

The Inverted Matrix frequent pattern mining or association rule algorithms are sets of algorithms with the purpose of mining large transactional databases with minimal candidacy generation and reducing the effects of superfluous work. These algorithms are divided among the two phases of the mining process namely the pre-processing in which the Inverted Matrix is built and the mining phase in which the discovery of frequent patterns occurs.

3.6.1 Building the Inverted Matrix

The Inverted Matrix is a disk-based data layout that is made of two parts: the index and the transactional array. The index contains the items and their respective frequency. The transactional array is a set of rows in which each row is associated with one item in the index part. Each row is made of pairs representing pointers, where each pair holds 2 information: the physical address in the index part of the next item in the same transaction, and the physical address in the row of the next item in the same transaction. Building the Inverted Matrix is done in two passes of the database during the pre-processing phase. The first pass scans the whole database to find the frequency of each item. The item list is then ordered in ascending order according to their frequency. Pass two of the database reads each transaction from the database and orders it also into ascending order based on the frequency of each item. In the index part, the location of the first item in the transaction is sought and an entry to its transactional array is added that holds the location of the next item in this transaction. For the second item the same process occurs, in which an entry in the transactional table of the second item is added to hold the location of the third item in the transaction. The same process is repeated for all items in this transaction. The following transaction is read next and the same occurs for all its items. This process repeats for all transactions in the database. Algorithm 3 depicts the steps needed to build the Inverted Matrix.

Algorithm 3 Inverted Matrix (IM) Construction

Input: Transactional Database ($\mathcal{D}$).
Output: Disk Based Inverted Matrix.

Pass I:

Scan $\mathcal{D}$ to identify unique items and determine their frequencies.
Sort the list of items in ascending order of their frequency.
Create the index part of the IM using the sorted list.

Pass II

while there is still a transaction T in the database ($\mathcal{D}$) **do**
  Sort the items in the transaction T into ascending order according to their frequency
  while there are items s_i in the transaction **do**
    Add an entry in its corresponding transactional array row with 2- parameters
    (A) Location in index part of the IM of the next item s_{i+1} in T or *null* if s_{i+1} does not exist.
    (B) Location of the next empty slot in the transactional array row of s_{i+1}, *null* if s_{i+1} does not exist.
  end while
end while

3.6.2 Mining the Inverted Matrix

Association rule mining starts by defining the support level σ. Based on the given support, the algorithm finds all frequent patterns that occur more than σ. The objectives behind the Inverted Matrix mining algorithm are two fold: first, minimizing the candidacy generation; second, eliminating the superfluous scans of non frequent items. To accomplish this, a *support border* is defined. This border indicates where to slice the Inverted Matrix to gain direct access to those items that are frequent. In other words, the border is the first item in the index of the Inverted Matrix that has a support greater or equal to σ. In our previous example in Section 3.4, the *support border* is located above item C since $\sigma > 4$.

For Each item $\mathcal{I}$ in the index of the slice of the inverted matrix is considered at a time starting from the least frequent, a COFI-tree for $\mathcal{I}$ is built by following the chain of pointers in the transactional array of the Inverted Matrix. This $\mathcal{I}$-COFI-tree is mined branch by branch starting with the node of the most frequent item and going upward in the tree to identify candidate frequent patterns containing $\mathcal{I}$. A list of these candidates is kept and updated with frequencies of the branches where they occur. Since a node could belong to more than one branch of the tree, a participation count is used to avoid re-counting items and patterns. Algorithm 4 presents the steps needed to generate the COFI-trees and mine them.

In our previous example in Table 3.2, if σ is greater than 4 then the first frequent item will be item C at location 16 in the index part of the Inverted Matrix. The first element in the transactional array for item C denotes that it shares the same transaction with the item at location 17 which is E. At location (17,1) we find that the other item A at location 18, shares with them the same transaction. From this, the first child node of C is created holding an entry for item E and another child node from E is created holding an entry for item A. The frequency of all these items are set to 1 and their participation is set to 0. The second entry of the transactional array of item C is (17,2), and at location (17,2) we find an entry of (18,2). This means that items E, and A also share another transaction with item C. Since entries for these items have already been created in the same order, then there will be no need to create new nodes as we will only increment their frequencies. By scanning all entries for item C with their chain, we can build the C-COFI-tree as in Figure 3.10A. Methods in Algorithm 2 are applied on the C-COFI-tree to generate all frequent patterns related to C, which are CE:5, CA:5, and CEA:5. The C-COFI-tree can be released at this stage, and its memory space can be used for the next tree.

The same process happens for the next frequent item that is at location 17 (item E). Figure 3.10.B presents its COFI-tree which generates the frequent pattern EA:6.

Algorithm 4 Creating and Mining COFI-trees

Input: Inverted Matrix (IM) and a minimum support threshold σ. **Output:** Full set of frequent patterns.

```
Frequency_Location = Apply binary search on the index part of the IM to find the Location of the first frequent item based on σ.
while Frequency_Location < IM_Size do
  A = Frequent item at location (Frequency_Location)
  A_Transactional = The Transactional array of item A
  Create a root node for the (A)-COFI-tree with both frequency.count and participation.count = 0
  Index_Of_TransactionalArray = 0
  while Index_Of_TransactionalArray < Frequency of item A do
    B = item from Transactional array at location (Index_Of_TransactionalArray)
    Follow the chain of item B to produce sub-transaction C
    Items on C form a prefix of the (A)-COFI-tree.
    if the prefix is new then
      frequency.count= 1 and participation.count=0 for all nodes in the path
    else
      Adjust the frequency.count of the already exist part of the path.
    end if
    Adjust the pointers of the Header list if needed
    Increment Index_Of_TransactionalArray
  end while
  MineCOFI-tree (A)
  Release (A) COFI-tree
  Increment Frequency_Location //to build the next COFI-tree
end while
```

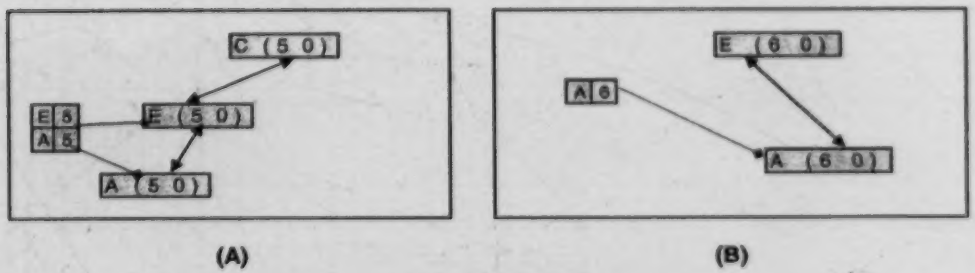


Figure 3.10: COFI-trees (A) Item C, (B) Item E

3.7 Inverted Matrix: Experimental Evaluations

To test the efficiency of the Inverted Matrix approach, we conducted experiments with settings identical to the ones described in Section 3.3.

Figure 3.11 and Figure 3.12 illustrate the comparative results obtained with *Apriori*, *FP-Growth* and the Inverted Matrix. *Apriori* failed to mine the 5 million transactional database and FP-Tree could not mine beyond the 5 million transaction mark. The Inverted Matrix, however, demonstrates good scalability as this algorithm mines 25 million transactions in 2731s. None of the tested algorithms, or reported results in the literature reaches such a big size.

2100	N/A	N/A	N/A	N/A	N/A
907	2401	N/A	N/A	N/A	N/A
430	730	1280	1830	2200	2731

Figure 3.11: Time needed in seconds to mine different transaction sizes

To test the behaviour of the Inverted Matrix vis-à-vis different support thresholds, a set of experiments was conducted on a database size of one million transactions, with 10,000 items and an average transaction length of 24 items. The matrix was built in about 763 seconds and it occupied a size of 109MB on the hard drive. The original transactional database with a horizontal layout uses 102MB. The mining process tested different support levels, which are 0.0025% that revealed almost 125K frequent patterns, 0.005% that revealed nearly 70K frequent patterns, 0.0075% that generated 32K frequent patterns and 0.01% that returned 17K frequent patterns. Figure 3.13 reports the time needed in seconds for each one of these runs. The results show that the Inverted Matrix algorithm outperforms both *Apriori* and *FP-Growth* algorithms in all cases. Figure 3.14 depicts the results of Figure 3.13. It is true that there was an overhead cost which was not recorded in Figure 3.13, namely the cost of building the Inverted Matrix. In this particular reported result we meant to focus on the actual mining time. In order to reduce the creation time we could build the Inverted Matrix in main-memory if the space is enough, in other words if the transactional database is small enough. This is not our goal as we focus on mining extremely large cases in which transactions are assumed not to fit in Main-Memory.

The Inverted Matrix is built only once and used to mine with four different support thresholds. The total execution time needed for *FP-Growth* to mine these four cases is 15607s, while *Apriori* needed 22500s, and the Inverted Matrix needed only 4540s, in addition to the 763s needed to build the matrix on disk. This makes the total execution time for the Inverted Matrix algorithms about 5303s, one third of the time needed by *FP-Growth*.

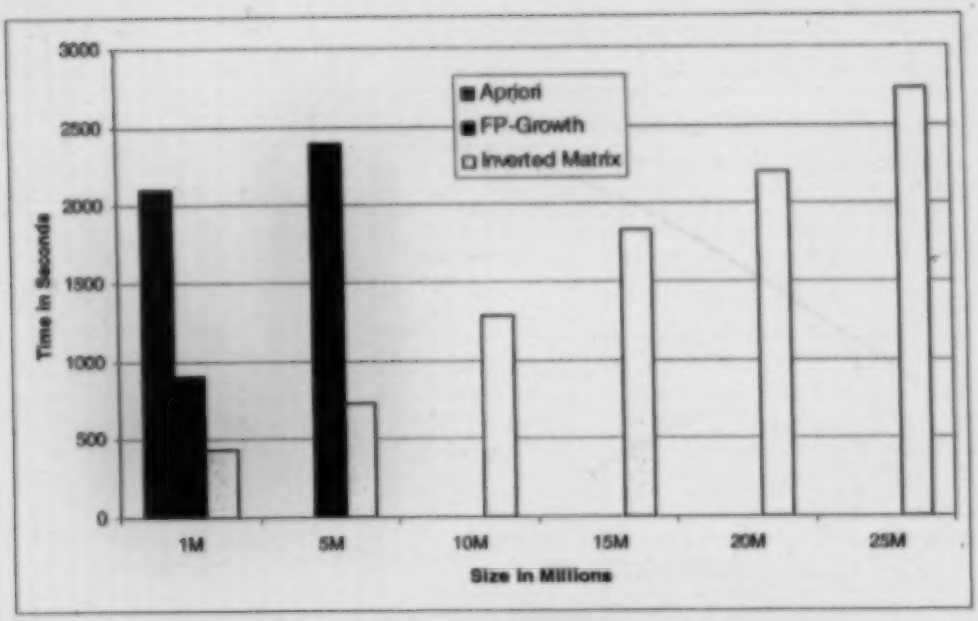


Figure 3.12: Time needed in seconds to mine different transaction sizes

Building the disk-based data structure once and mining it many times by using different supports, saves the overall execution time in comparison with other algorithms. This total time for all runs is illustrated in Figure 3.15. More time improvements could be achieved if more support levels are tested, amortizing the building time over many runs. Notice that given the highly interactive nature of most KDD processes, a "build-once-mine-many" approach is always desirable.

Note that the advantage that COFI with Inverted Matrix has over *FP-Growth* is still valid for FP-COFI since our FP-COFI also has to build the FP-tree for each run.

	11300	6200	2900	2100
	6800	5600	2300	907
	2300	1030	780	430

Figure 3.13: Time needed to mine 1M transactions with different supports level

3.8 Summary

Finding scalable algorithms for association rule mining in extremely large databases is the main goal of our research. To reach this goal, we propose a new set of algorithms that uses

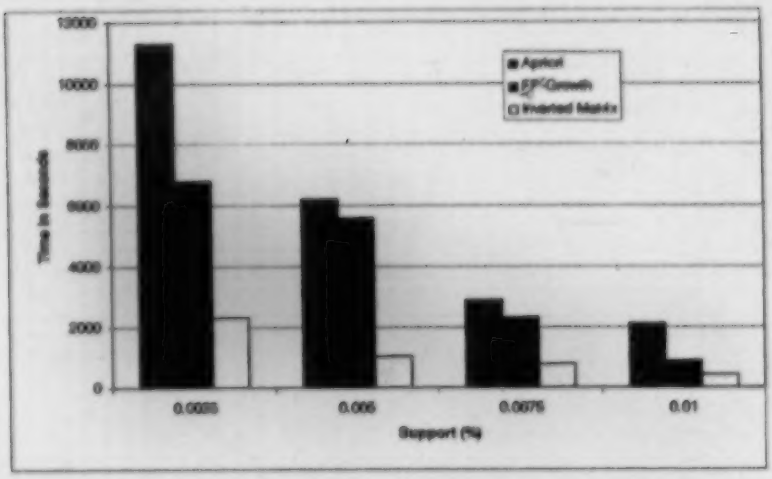


Figure 3.14: Time needed to mine 1M transactions with different supports levels

the disk to store the transactions in a special layout called Inverted Matrix. It also uses the memory to interactively mine relatively small structures called COFI-trees that can also be used with FP-Trees. The experiments we conducted showed that our algorithms are scalable to mine tens of millions of transactions, if not more. Our study reinforces that in mining extremely large transactions we should neither work on algorithms that build huge memory data structures, nor on algorithms that scan the massive transactions many times.

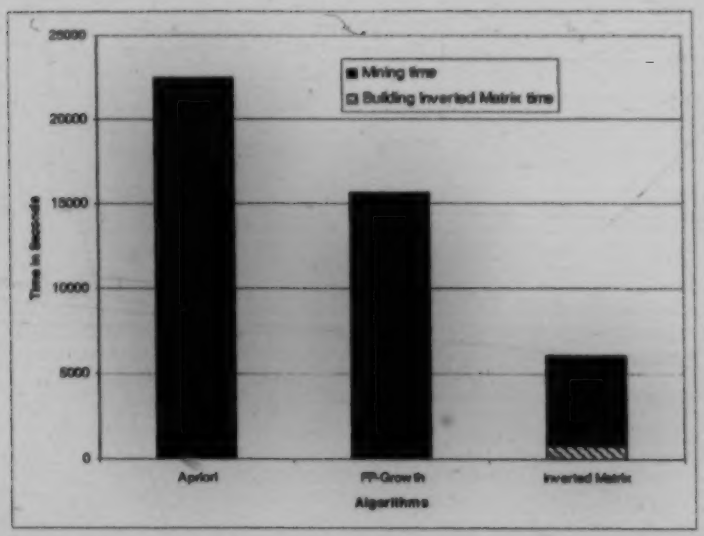


Figure 3.15: Accumulated time needed to mine 1M transactions using four different support levels, including the preprocessing phase

Chapter 4

Leap-Traversal approach

The fun is in the search, not the finding.
– German Proverb

A great leap gives a great shake.
– Spanish Proverb

Regardless of the frequent patterns to discover, be it the full frequent patterns or the condensed ones, closed or maximal, the strategy always includes the traversal of the lattice of candidate patterns. We study the existing depth versus breadth traversal approaches for generating candidate patterns and propose in this chapter a new traversal approach that jumps in the search space among only promising nodes. Our leaping approach avoids nodes that would not participate in the answer set and reduces drastically the number of candidate patterns. We use this approach to efficiently pinpoint maximal or closed patterns at the border of the frequent patterns in the lattice and collect enough information in the process to generate all subsequent patterns.

Discovering frequent patterns is a fundamental problem in data mining. Efficiently tacking this problem is by no means solved and remains a major challenge, particularly for extremely large databases. The idea behind the algorithms to solve this problem is the identification of a relatively small set of candidate patterns, and counting those candidates to keep only the frequent ones. The fundamental difference between the algorithms lies in the strategy to traverse the search space and to prune irrelevant parts. For frequent itemsets, the search space is a lattice connecting all combinations of items between the empty set and the set of all items. Regardless of the pruning techniques, the sole purpose of an algorithm is to reduce the set of enumerated candidates to be counted. The strategies adopted for traversing the lattice are always systematic, either depth-first or breadth-first, traversing the space of itemsets either top-down or bottom-up. Among these four strategies, there is never a clear winner, since each one either favours long or short patterns, thus heavily relying on the transactional database at hand. Our primary motivation here is to find a new

traversal method that neither favours nor penalizes a given type of dataset, and at the same time allows the application of lattice pruning for the minimization of candidate generation. Moreover, while discovering frequent patterns can shed light on the content and trends in a transactional database, the discovered patterns can outnumber the transactions themselves, making the analysis of the discovered patterns impractical and even useless. New attempts toward solving such problems are made by finding the set of maximal frequent patterns [1, 8, 16, 44, 45], where a frequent itemset is said to be maximal if there is no other frequent itemset that subsumes it. While we can derive the set of all frequent itemsets directly from the maximal patterns, their support cannot normally be obtained without counting with an additional database scan. The basic idea of Leap-Traversal we present herein is to traverse the itemset lattice in search of frequent maximals first. Our approaches collect enough information in the process to be able to generate all frequent patterns with their exact support from this set of maximals without having to perform an additional data scan. The data structure used to perform this is presented herein. The same Leap-Traversal idea can be extended to directly generate the set of closed patterns.

In this work we study the new traversal approach, called Leap-Traversal, and integrate it in two strategies: one that mines a variation of FP-Tree called Headerless FP-Tree and the other one that partitions using COFI-trees. While these approaches are not particularly competitive with small datasets, we show superior performance of HFP-Leap and COFI-Leap over other approaches with very large datasets (real and synthetic).

4.1 Traversal Approaches

Existing frequent pattern mining algorithms use either breadth-first-search or depth-first-search strategies to find candidates that will be used to determine the frequent patterns. Those strategies have been described in Section 2.3

It is true that many algorithms have been published for enumerating and counting frequent patterns, and yet most algorithms still use one of the two traversal strategies (depth-first versus breadth-first) in their search. They differ only in pruning techniques and structures used. No work has been done to find new traversal strategies, such as greedy ones, or best first, etc. We need a new greedy approach that jumps in the lattice searching for the most promising nodes and based on these nodes, generates the set of maximals, closed, or all frequent patterns.

4.1.1 Leap-Traversal Approach: Candidate Generation versus Maximal Generation

Most frequent itemset algorithms follow the candidate generation first approach, where candidate items are generated first and only the candidate with support higher than the

predefined threshold are declared as frequent while others are omitted. One of the main objectives of the existing algorithms is to reduce the number of candidate patterns. In this chapter, we propose a new approach to traverse the search space for frequent patterns that is based on finding two things: the set of maximal patterns, and a data-structure that encodes the support of all frequent patterns that can be generated from the set of maximal frequent patterns.

Since maximal patterns alone do not suffice to generate the subsequent patterns, the data structure we use keeps enough information about frequencies to counter this deficiency. The basic idea behind the Leap-Traversal approach is that we try to identify the frequent pattern border in the lattice by marking some particular patterns (called later *Frequent-Path-Bases*). Simply put, the marked nodes are those representing complete sub-transactions of frequent items.

How these are identified and marked will be discussed later. If those marked patterns are frequent, they belong to the border explained earlier in Section 2.3.1 (i.e. they are potential maximal); otherwise, their subsets could be frequent, and thus we jump in the lattice to patterns derived from the intersection of the infrequent marked patterns in the anticipation of identifying the frequent pattern border. The intersection comes from the following intuition: if a marked node is not a maximal, a subset of it should be maximal. However, rather than testing all its descendants, to reduce the search space, we look at descendants of two non frequent marked nodes at a time, hence, the pattern intersection. The process is repeated until all currently intersected marked patterns are frequent and hence the border is found.

Before we explain the Leap-Traversal approach in detail, let us define the *Frequent-Path-Bases* (FPB). These are some particular patterns in the itemset lattice that we mark and use for our traversal. An FPB, if frequent, could be a maximal. If infrequent, one of its subsets could be frequent and maximal. A *Frequent-Path-Base* for an item A, called *A-Frequent-Path-Base*, is a set of frequent items that has the following properties:

1. At maximum one A-FPB can be generated from one transaction.
2. All frequent items in an *A-Frequent-Path-Base* have support greater than or equal to the support of A;
3. Each A-FPB represents items that physically occur in the database with item A.
4. Each A-FPB has its branch-support, which represents the times A-FPB occurs in the database not as subset of other FPBs. In other words, the branch support of a pattern is the number of transactions that consist of this pattern, not the transactions that include this pattern along with other frequent items. The branch support is always less or equal to the support of a pattern.

As an example for the Leap-Traversal, assuming we have an oracle to generate for us the *Frequent-Path-Bases* from the token database in Figure 4.1, the same figure, with minimum support equal to 2, illustrates the process. With the initial FPBs ABC, ABCD, ACDE and DE (given by our hypothetical oracle), we end up testing only 10 candidates to discover 13 frequent patterns: two were tested unnecessarily (ABCD and ACDE) but 5 patterns were directly identified as frequent without even testing them: AB, AC, AD, BC, CD. From the initially marked nodes, ABC and DE are found to be frequent, but ABCD and ACDE are not. The intersection of those two nodes yields ACD. This newly marked node is found to be frequent and thus, maximal. From the maximals ACD, DE and ABC, we generate all the subsequent patterns, some even without testing (AB, AC, AD, BC and CD). The supports of these patterns are calculated from their superset FPBs. For example, AC has the support of 4 since ABC occurs (alone) twice, ABCD and ACDE each occur alone once. Section 2.3.1 presents the same example using both breadth-first versus depth-first search, where the former search tests 18 candidates to finally discover the 13 frequent patterns and the later one tests 23 candidates.

Frequent-Path-Bases that have support greater than the predefined support (i.e. frequent patterns) are put aside as they are already known to be frequent and all their subsets are also known to be frequent. Only infrequent ones participate in the Leap-Traversal approach, which consists of intersecting non frequent FPBs to find a common subset of items shared among the two intersected patterns. The support of this new pattern is found as follows without revisiting the database: the support of Y where $Y = FPB1 \cap FPB2$, is the summation of the branch support of all FPBs that are superset of Y. For example, if we have only two *Frequent-Path-Bases* ABCD: 1, and ABEF: 1, by intersecting both, FPBs we get AB that occurs only once in ABCD and once in ABEF, which means it occurs twice in total. By doing so, we do not have to traverse any candidate pattern of size 3 as we were able to jump directly to the first frequent pattern of size 2, which can be declared defacto as a maximal pattern, hence the name Leap-Traversal. Consequently, all its subsets are also frequent, which are A and B with support of 2 as they occur only once in each of the *Frequent-Path-Bases*.

The Leap-Traversal approach starts by building a lexicographic tree of intersections among the *Frequent-Path-Bases*. It is a tree of possible intersections between existing FPBs ordered in a lexicographic manner. Assume we have 6 FPBs A, B, C, D, E, and F then Figure 4.2 depicts the lexicographic tree of intersections between these pattern bases. The size of this tree is relatively big as it has a depth equal to the number of FPBs, which is 6 in our case. It is also unbalanced to the left since intersection is commutative. The number of nodes for this tree is equal to 52, as the number of nodes in a lexicographic tree is equal to $\sum_{i=1}^{n} \binom{n}{i}$ where n is the number of *Frequent-Path-Bases*. It is obvious that the more FPBs

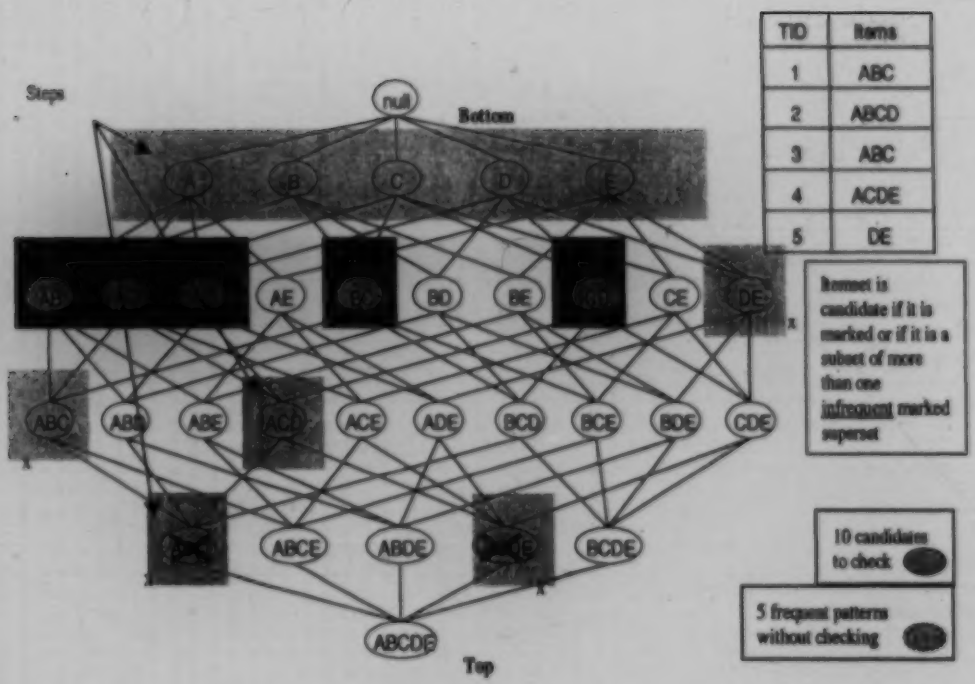


Figure 4.1: Leap-Traversal

we have, the larger the tree becomes. Thus, pruning this tree plays an important role in having an efficient algorithm.

Four pruning techniques can be applied to the lexicographic tree of intersections. These pruning strategies can be explained by the following theorems:

Theorem 1: $\forall X, Y \in FPBs$ *ordered lexicographically, if* $X \cap Y$ *is frequent then there is no need to intersect any other elements that have* $X \cap Y$*, i.e, all children of* $X \cap Y$ *can be pruned.*

Proof: $\forall A, X, Y \in FPBs$, $A \cap X \cap Y \subset X \cap Y$. If $X \cap Y$ is frequent then $A \cap X \cap Y$ is also frequent (*apriori* property) as a subset of a frequent pattern is also frequent.

Example: Assuming that the support threshold in all of the following examples is 5 then from Figure 4.2, we can find that $A \cap B$ generates the pattern 1,3,4,5,9 that has a support of 5 as it is a subset of A, B, C, D, F where all of them have support-branch equal to 1. Since $A \cap B$ is frequent then all its subsets are also frequent and can be pruned from the tree. In other words, there is no need to do the intersections again.

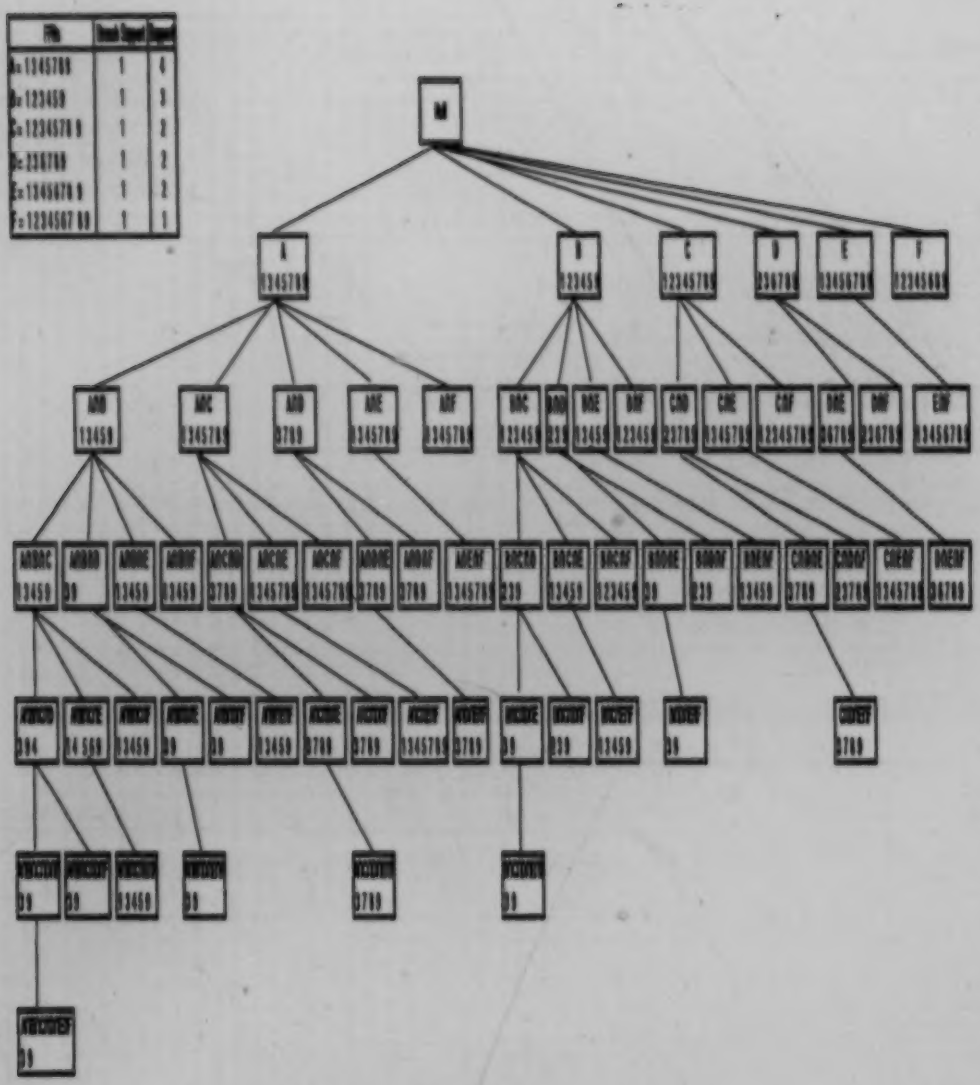


Figure 4.2: Lexicographic tree of intersections

Theorem 2: $\forall X, Y, W \in FPBs$ *ordered lexicographically, if* $X \cap Y = X \cap W$ *then there is no need to explore any children of* $X \cap Y$.

Proof: There exist a FPB Z which is between Y and W, where Z is either smaller lexicographically or equal to W. To prove this theorem, we need to show that any child $X \cap Y \cap Z$ of $X \cap Y$ is repeated under another pattern $X \cap Z$ that always exists, Z could be W. Since $X \cap Y = X \cap W$, then $X \cap Y \cap Z = X \cap Z \cap W$ (intersection is commutative)

Example: From Figure 4.2, $A \cap C$ produced the same pattern such as $A \cap E$, which is 1, 3, 4, 5, 7, 8, 9. All children of $A \cap C$ are already repeated at the right of $A \cap C$. $A \cap C \cap D$ is repeated in $A \cap D \cap E$, $A \cap C \cap E$ and $A \cap C \cap F$ are repeated in $A \cap E \cap F$. From this, we can prune $A \cap C$ nodes and consequently we will not have to process the intersections in its children nodes.

Theorem 3: $\forall X, Y, Z \in FPBs$ *ordered lexicographically, if* $X \cap Y \subset X \cap Z$ *then we can ignore the subtree* $X \cap Y \cap Z$.

Proof: Assume we have $X, Y, Z \in FPBs$, Since $X \cap Y \subset X \cap Z$ then $X \cap Y \cap Z = X \cap Y$. This means we do not get any additional information by intersecting Z with $X \cap Y$. Thus, the subtree under $X \cap Y$ suffices.

Example: From Figure 4.2, $A \cap D$ is a subset of $A \cap F$. Knowing this, we can conclude that there is no need to generate and test the $A \cap D \cap F$ node.

Theorem 4: $\forall X, Y, Z \in FPBs$, *if* $X \cap Y \supset X \cap Z$ *then we can ignore the subtree of* $X \cap Z$ *as long* $X \cap Z$ *is not frequent.*

Proof: Following the proof of Theorem 3, we can conclude that $X \cap Z$ is included in $X \cap Y$.

Example: In Figure 4.2, $B \cap C \supset B \cap D$. From this, we can find that we can prune the pattern generated from $B \cap D$ without losing any valuable information as we were able to regenerate the same pattern from $B \cap C \cap D$.

Lemma 1: At each level of the lexicographic tree of intersections, consider each item as a root of a new subtree:

(A) Intersect the siblings for each node with the root

(B) If the node does not have an empty itemset and this itemset is not frequent then we

can prune that node.

Proof: Assume we have $X, Y, Z \in FPBs$, if X is a parent node and $X \cap Y \cap Z$ exists and is not frequent then any superset for this intersected node is also not frequent (*apriori* property). That is why any intersection of X with any other item is also not frequent.

Example: The right children for node D for example are E, F. By intersecting all of them together, we can find that a new pattern 1,3,4,5,7,8,9 can be generated that has a support of 4. Knowing this, all supersets of this pattern will also be infrequent. Hence node D can be pruned as all its children will also be non frequent.

4.1.2 Heuristics used for building and traversing the lexicographic tree

Heuristic 1: The lexicographic tree of intersections of FPBs needs to be ordered. Four ways of ordering could be used: order by support, support branch, pattern length, and random. Ordering by support yields the best results, as intersecting two patterns with high support in general would generate a pattern with higher support than intersecting two patterns with lower support. Ordering the tree by assigning the high support nodes at the left increases the probability of finding early frequent patterns in the left and by using Theorem 1, a larger subtree can be pruned.

Heuristic 2: The second heuristic deals with the traversal of the lexicographic tree. The breadth-traversal of the tree is better than the depth-traversal. This observation can be justified by the fact that the goal of the lattice Leap-Traversal approach is to find the maximal patterns, which means finding longer patterns early is the goal of this approach. Thus, by using the breadth-first approach on the intersection tree, we detect and test the longer patterns early before applying too many intersections that usually lead to smaller patterns.

4.2 Tree Structures Used

The Leap-Traversal approach can be applied by using two strategies: Headerless Frequent Pattern Leap (HFP-Leap) or COFI-Leap, both of which will be explained later in this chapter. In our work, we mainly adopted COFI-Leap for the sequential implementation and the HFP-Leap for the parallel one. The reason for this adoption is explained in Chapter 6.

Algorithm 8 that performs the actual Leap-Traversal to find maximal patterns is called from both strategies. We will first present the idea behind HFP-Leap and then show the use of COFI-trees to perform the same type of jumps in the lattice.

In the first strategy, the Leap-Traversal approach we discuss consists of two main stages: the construction of a Headerless Frequent Pattern tree (HFP-Tree); and the actual mining for this data structure by building the tree of intersected patterns.

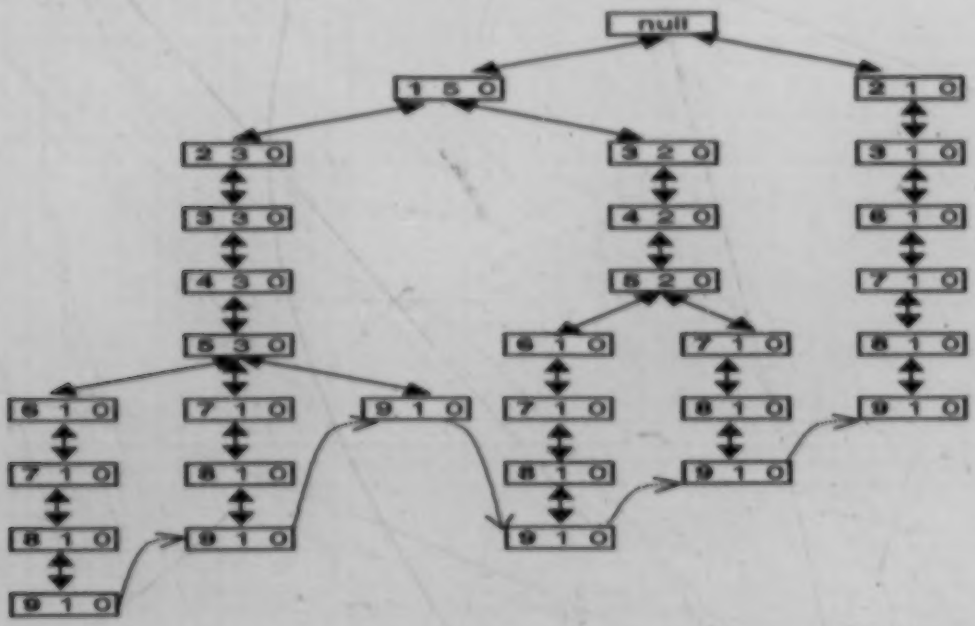


Figure 4.3: Headerless FP-Tree: An Example.

4.2.1 Construction of the Headerless Frequent Pattern Tree

The goal of this stage is to build the compact data structure similar to the Frequent Pattern Tree, which is a prefix tree representing sub-transactions pertaining to a given minimum support threshold. This data structure compressing the transactional data was contributed by Han et al. [48]. The tree structure we use, called HFP-Tree, is a variation of the original FP-Tree. However, we will quickly introduce the original FP-Tree before discussing the differences with our data structure. The construction of the FP-Tree is done in two phases, where each phase requires a full I/O scan of the database. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

After the enumeration of the items appearing in the transactions, infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called a header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. The actual frequent pattern tree is built in the second phase. This phase requires a second complete I/O scan of the database. For each

transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Tree.

Each ordered sub-transaction is compared to the prefix tree starting from the root. If there is a match between the prefix of the sub-transaction and any path in the tree starting from the root, then the support in the matched nodes is simply incremented. Otherwise nodes are added for the items in the suffix of the transaction to continue a new path, each new node having a support of one. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

Our tree structure is the same as the FP-Tree except for the following differences. We call this tree Headerless-Frequent-Pattern-Tree or HFP-Tree.

1. We do not maintain a header table, as a header table is used to facilitate the generation of the conditional trees in the *FP-Growth* model. It is not needed in our Leap-Traversal approach;

2. We do not need to maintain the links between the same itemset across the different tree branches (horizontal links);

3. The links between nodes are bi-directional to allow top-down and bottom-up traversals of the tree;

4. All leaf nodes are linked together as the leaf nodes are the start of any pattern base and linking them helps the discovery of *Frequent-Path-Bases*;

5. In addition to *support*, each node in the HFP-Tree has a second variable called *participation*. *Participation* plays a similar role in the mining process as the *participation* counter in the COFI-tree in Section 3.2.1.

Basically, the support represents the support of a node, while participation represents, at a given time in the mining process, the number of times the node has participated in already counted patterns. Based on the difference between the two variables, *participation* and *support*, the special patterns called *Frequent-Path-Bases* are generated. These are simply the paths from a given node x, with participation smaller than the support, up to the root, (i.e. nodes that did not fully participate yet in frequent patterns). Figure 4.3 presents the Headerless FP-Tree for the same dataset used in Figure 4.2.

Algorithm 5 shows the main steps in our approach. After building the Headerless FP-Tree with 2 scans of the database, we mark some specific nodes in the pattern lattice using

FindFrequentPatternBases. Using the FPBs, the Leap-Traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice.

Algorithm 5 HFP-Leap: Leap-Traversal with Headerless FP-Tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns with their respective supports.

Scan D to find the set of frequent 1-itemsets $F1$
Scan D to build the Headerless FP-Tree HFP
$FPB \leftarrow$ FindFrequentPatternBases(HFP)
$Maximals \leftarrow$ FindMaximals(FPB, σ)
Output $Maximals$

Algorithm 6 shows how patterns in the lattice are marked. The linked list of leaf nodes in the HFP-Tree is traversed upward to find the unique paths representing sub-transactions. If frequent maximals exist, they have to be among these complete sub-transactions. The participation counter helps reuse nodes exactly as needed to determine the *Frequent-Path-Bases*. Figure 4.4 presents the steps needed to generate *Frequent-Path-Bases* from a HFP-Tree.

4.2.2 Construction of the COFI-trees

Algorithm 6 FindFrequentPatternBases: Marking nodes in the lattice

Input: HFP (Headerless FP-Tree) OR $A - COFI$.
Output: FPB (*Frequent-Path-Bases* with counts)

$ListNodesFlagged \leftarrow \emptyset$
Follow the linked list of leaf nodes in HFP
for each leaf node N **do**
  Add N to $ListNodesFlagged$
end for
while $ListNodesFlagged \neq \emptyset$ **do**
  $N \leftarrow$ Pop($ListNodesFlagged$) {from top of the list}
  $fpb \leftarrow$ Path from N to *root*
  fpb.branchSupport $\leftarrow N$.support $- N$.participation
  for each node P in fpb **do**
    P.participation $\leftarrow P$.participation $+$ fpb.branchSupport
    if P.participation $< P$.support AND $\forall c$ child of P, c.participation $= c$.support **then**
      add P in $ListNodesFlagged$
    end if
  end for
  add fpb in FPB
end while
RETURN FPB

A COFI-tree as described in Section 3.2 is a projection of each frequent item in the original FP-Tree [48] (not the Headerless FP-Tree). Each COFI-tree, for a given frequent item, presents the co-occurrence of this item with other frequent items that have more

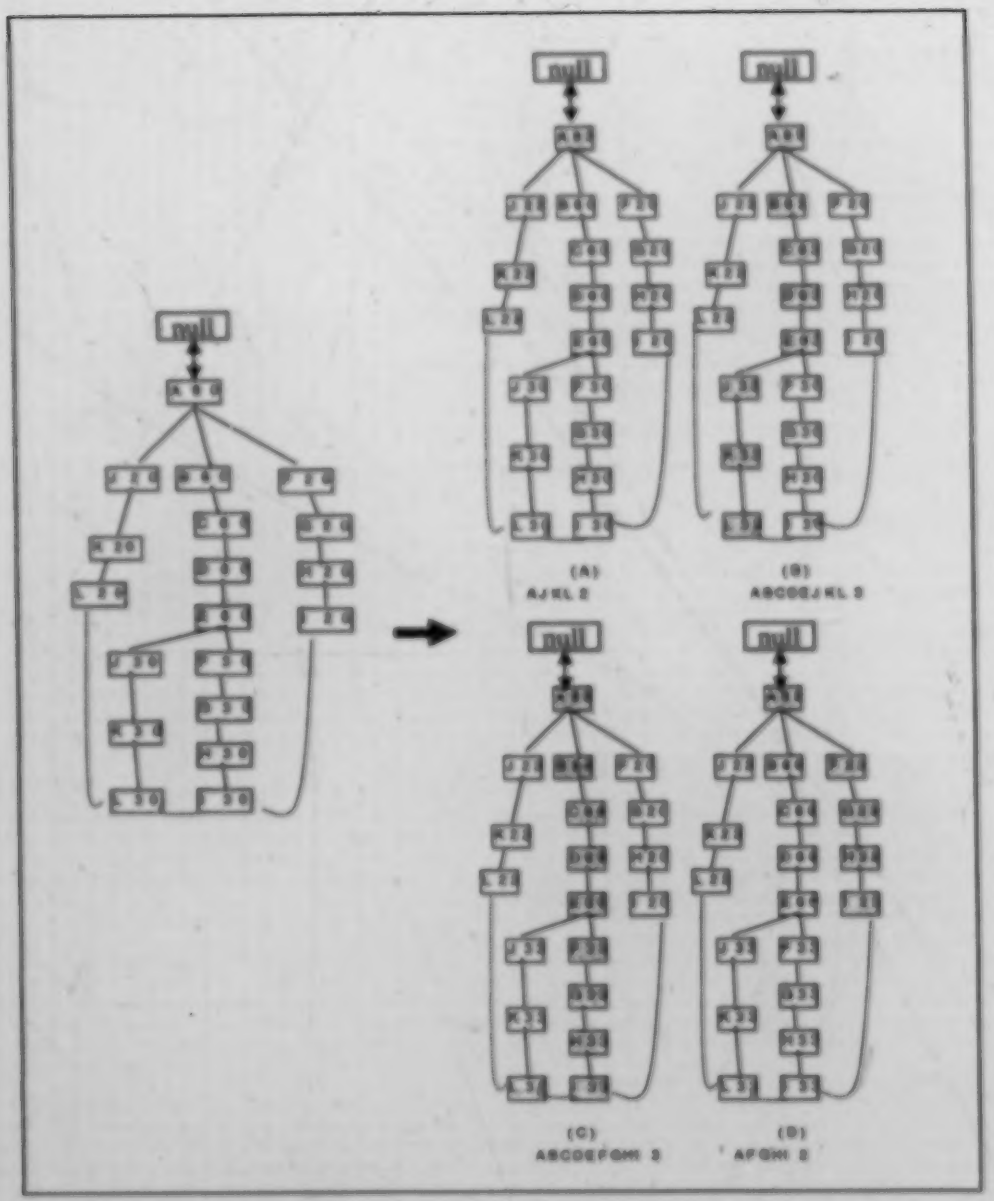


Figure 4.4: Steps needed to generate *Frequent-Path-Bases* from a HFP-Tree

support than itself. In other words, if we have 4 frequent items A, B, C, D where A has the smallest support, and D has the highest, then the COFI-tree for A presents co-occurrence of item A with respect to B, C and D, the COFI-tree for B presents item B with C and D. The COFI-tree for C presents item C with D. Finally, the COFI-tree for D is a root node tree. Each node in the COFI-tree, similar to the Headerless FP-Tree, has two main variables, support and participation. Participation indicates the number of patterns the node has participated in at a given time during the mining step. Based on the difference between these two variables, participation and support, *Frequent-Path-Bases* are generated. The COFI-tree has also a header table that contains all locally frequent items with respect to the root item of the COFI-tree. Each entry in this table holds the local support, and a link to connect its item with its first occurrences in the COFI-tree. A link list is also maintained between nodes that hold the same item to facilitate the mining procedure. *Frequent-Path-Bases* are generated from each COFI-tree alone using the same approach as the Headerless FP-Tree. One of the advantages of using COFI-trees over the Headerless FP-Tree is that we can skip building some COFI-trees during the mining process. This is due to the fact that before we build any COFI-tree, we check all its local frequent items. If all its items are a subset of an already discovered maximal pattern, then there is no need to build and mine this COFI-tree as all its sub-patterns are subsets of already discovered maximal patterns.

Algorithm 7 COFI-Leap: Leap-Traversal with COFI-tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns with their respective supports.

Scan D to find the set of frequent 1-itemsets $F1$
Scan D to build the FP-Tree $FP-TREE$
$A \leftarrow$ frequent item with least support
for $\forall A$ **do**
  Generate COFI-tree for A, $A-COFI$
  $FPB \leftarrow$ FindFrequentPatternBases($A-COFI$)
  $Maximals \leftarrow$ FindMaximals(FPB, σ)
  Output $Maximals$
  Clear $A-COFI$
  $A \leftarrow$ Next item with larger support if still exists
end for

Algorithm 7 shows the main steps in the COFI-Leap approach. After building the FP-Tree with 2 scans of the database, we create independent COFI-trees. In each of the COFI-trees, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases*. Using the FPBs, the Leap-Traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice.

4.2.3 Actual mining of *Frequent-Path-Bases*: The Leap-Traversal approach

Algorithm 8 is the actual Leap-Traversal to find maximals using FP-Trees generated all at one time using the Headerless FP-Tree or in chunks using COFI-tree approach. It starts by listing some candidate maximals stored in *PotentialMaximals* which is initialized with the *Frequent-Path-Bases* that are frequent. All the non frequent FPBs are used for the jumps of the lattice Leap-Traversal. These FPBs are stored in the list *List*, and intermediary lists *NList* and *NList2* will store the nodes in the lattice that the intersection of FPBs would point to, or in other words, the nodes that may lead to maximals. The nodes in the lists have two attributes: *flag* and *startpoint*. For a node n, *flag* indicates that a subtree in the intersection tree should not be considered starting from the node n. For example, if node $(A \cap B)$ has a flag C, then the subtree under the node $(A \cap B \cap C)$ should not be considered. For a given node n, *startpoint* indicates which subtrees in the intersection tree, descendants of n, should be considered. For example, if a node $(A \cap B)$ has the startpoint D, then only the descendants $(A \cap B \cap D)$ and so on are considered, but $(A \cap B \cap C)$ is omitted. Note that $ABCD$ are ordered lexicographically. At each level in the intersection tree, when *NList2* is updated with new nodes, the theorems are used to prune the intersection tree. In other words, the theorems help avoid useless intersections (i.e. useless maximal candidates). The same process is repeated for all levels of the intersection tree until there is no other intersections to do (i.e. *NList2* is empty). At the end, the set potential maximals is cleaned by removing subsets of any sets in *PotentialMaximals*.

It is obvious in the Leap-Traversal approach that superset checking and intersections plays an important role. We found that the best way to work with this is by using the bit-vector approach where each frequent item is represented by one bit in a vector. In this approach, intersection is nothing but applying the AND operation between two vectors, and subset checking is nothing but applying the AND operation between two vectors. If $A \cap B = A$ then A is a subset of B.

4.3 Leap-Traversal Applications

The Leap-Traversal approach can also be applied to find the set of closed and all frequent patterns. It can also be used to push constraints during the mining process as explained in Chapter 5. Its parallelization is also shown to be relatively easy as illustrated in Chapter 6, due to the fact that it was originally designed with parallelization in mind.

4.3.1 Closed and All Frequent Patterns

The main goal of the Leap-Traversal approach is to find the set of maximal patterns. From this set we can generate all the subset patterns where all subsets are the set of all frequent

Algorithm 8 FindMaximals: The actual Leap-Traversal

Input: FPB (Frequent Pattern Bases); σ (Support threshold).
Output: $Maximals$ (Frequent Maximal patterns){which FPBs are maximals?}
$List \leftarrow FPB$ and $PotentialMaximals \leftarrow \emptyset$
for each i in $List$ **do**
  Find support of i {using branch supports}
  if support(i) $> \sigma$ **then**
    Add i to $PotentialMaximals$
    Remove i from $List$
  end if
end for
Sort $List$ based on support
$NList \leftarrow List$
$NList2 \leftarrow \emptyset$
$\forall i \in NList$ initialize i.flag $\leftarrow NULL$ AND i.startpoint $\leftarrow$ index of i in $NList$
while $NList \neq \emptyset$ **do**
  {Intersections of FPBs to select nodes to jump to}
  for each i in $NList$ **do**
    $g \leftarrow$ Intersect(i, j) {where $j \in List$ AND $i \ll j$ (in lexicographic order) AND not j.flag}
    g.startpoint $\leftarrow j$
    Add g to $NList2$
  end for
  {Pruning starts here}
  for each i in $NList2$ **do**
    Find support of i {using branch supports}
    if support(i) $> \sigma$ **then**
      Add i to $PotentialMaximals$
      Remove all duplicates or subsets of i in $NList2$; Remove i from $NList2$ {Theorem 1}
    else
      Remove all duplicates of i in $NList2$ except the most right one ; Remove i from $NList2$ {Theorem 2}
      Remove all non frequent subsets of i from $NList2$ {Theorem 4}
      if $\exists j \in NList2$ AND $j \supseteq i$ **then**
        i.flag $\leftarrow j$ {Theorem 3}
      end if
      for all j in $List$ **do**
        if $j \gg i$.startpoint (in lexicographic order) **then**
          $n \leftarrow$ Intersect(i, j)
          Find support of n {using branch supports}
          if support(n) $< \sigma$ **then**
            Remove i from $NList2$ {Lemma 1}
          end if
        end if
      end for
    end if
  end for
  $NList \leftarrow NList2$
  $NList2 \leftarrow \emptyset$
end while
Remove any x from $PotentialMaximals$ if ($\exists M \in PotentialMaximals$ AND $x \subset M$)
$Maximals \leftarrow PotentialMaximals$
RETURN $Maximals$

Algorithm 9 GeneratePatterns: Extending the maximals

Input: *FPB* (*Frequent-Path-Bases*); *Maximals* (Set of frequent maximals); *Type* (Closed or All).
Output: *Patterns* (Either closed or all frequent patterns with their supports)

```
for each M in Maximals do
  FP ← Generate all sub-patterns of M
end for
for each p in FP do
  p.support = ∑∀X.branchSupport {Where X ∈ FBS AND p ⊂ X}
  add p in Patterns
end for
RETURN Patterns
```

patterns, and some of them are the set of closed patterns. The only challenge in this process is to compute the support of these patterns. The computation for the support for these patterns is encoded in the branch support of the existing FPBs already generated either from the HFP-Tree or COFI-trees, where the support of any generated pattern is the summation of the branch support of all its supersets of FPBs.

As can be seen in Algorithm 9, all relevant patterns are generated from the set of maximals. Using the definition of maximals, all subsets of a maximal are de facto frequent. Once their support is computed using the branch support of FPBs as described above, pinpointing closed patterns is simply done using the definition of closed itemsets (i.e. no other frequent pattern subsumes it and has the same support).

4.4 Performance Evaluations

To evaluate our Leap-Traversal approach, we conducted a set of different experiments using both approaches HFP-Leap and COFI-Leap. First, we measured their effectiveness compared to other algorithms when mining relatively small datasets. We also compared our algorithms with some of the state-of-the-art algorithms solely to discover the maximal patterns, in terms of speed, memory usage and scalability.

For mining Maximal Frequent Itemsets (MFIs), Depth-Project [1] was shown to achieve more than one order of magnitude speedup over MaxMiner [8]. MAFIA [16] was shown to outperform DepthProject by a factor of 3 to 5. Gouda and Zaki presented GENMAX that has been described in their work [44] as the current best method to mine the set of exact MFIs. They also claim that MAFIA is the best method for mining the superset of all MFIs.

The contenders we tested against are MAFIA [16], FPMAX [45] and GENMAX [44]. MAFIA was shown to outperform MaxMiner [8] and Depth-Project [1] for mining maximal itemsets. FPMAX is an extension of the *FP-Growth* [48] approach. We used an enhanced code of FPMAX that won the FIMI-2003 [43] award for best frequent mining implemen-

tation. The implementations of these algorithms were all provided to us by their original authors or downloaded from the FIMI repository [43]. We used the latest version of MAFIA that does not need a post-pruning step and generates directly the set of exact MFIs. We did not use the Inverted Matrix in our implementations of COFI-LEAP and HFP-Leap as Inverted Matrix has an advantage only when interactive mining is intended. All our experiments were conducted on an IBM P4 2.6GHz with 1GB memory running Linux 2.4.20-20.9 Red Hat Linux release 9. Timing for all algorithms includes the pre-processing cost such as horizontal to vertical conversions for both GenMax and MAFIA. The time reported also includes the program output time. We tested these algorithms using both real and synthetic datasets. All experiments were forced to stop if their execution time reached our wall time of 5000 seconds. We made sure that all algorithms reported the same exact set of frequent itemsets on each dataset (i.e. no false positives and no false negatives).

4.4.1 Mining Relatively Small Databases, Real and Synthetic

The first set of experiments we conducted mined real datasets such as Plant-Protein and retail. Plant-Protein data is a very dense dataset with about 3000 transactions using more than 7000 items (subsequence of amino-acids). The transactions represent plant proteins extracted from SWISS-PROT. In these experiments we found that FPMAX is almost always the winner in terms of speed. On the other hand, we also found that these algorithms (except Leap approach algorithms) use extremely large amount of memory, in spite of the fact that the tested database where in general small in terms of number of transactions. All experiments on the Plant-Protein and retail database are depicted in Figures 4.5, and 4.8 for the time comparison ones, and in Figures 4.6, 4.7, 4.9, and 4.10 for the memory usage comparison ones. Each slice in these figures presents the percentage of memory space needed by each algorithm compared to the total memory space needed by all algorithms to perform the same task. In other words, the complete circle which presents 100% represents the cumulation of all memory space needed by all test algorithms. The slice presents a relative share of a particular algorithm, the smaller the share the better. Of all The above observation raised the following question, "How do these algorithms behave once they start mining extremely large databases?" To test this idea we experimented these algorithms on synthetic databases ranging from 5,000 transactions up to 50 million transactions, with dimensions ranging from 5000 items to 100,000 items.

In another set of experiments we tested other datasets, UCI datasets and synthetic ones, with one goal in mind: Finding the best mining algorithm. We tested all the enumerated algorithms using 4 databases made available by Goethals and Zaki [43]. These databases are *chess, mushroom, pumsb, and accidents.* The characteristics of these databases are described in Table 4.1. We have also generated synthetic datasets using Quest [51]. In these sets of

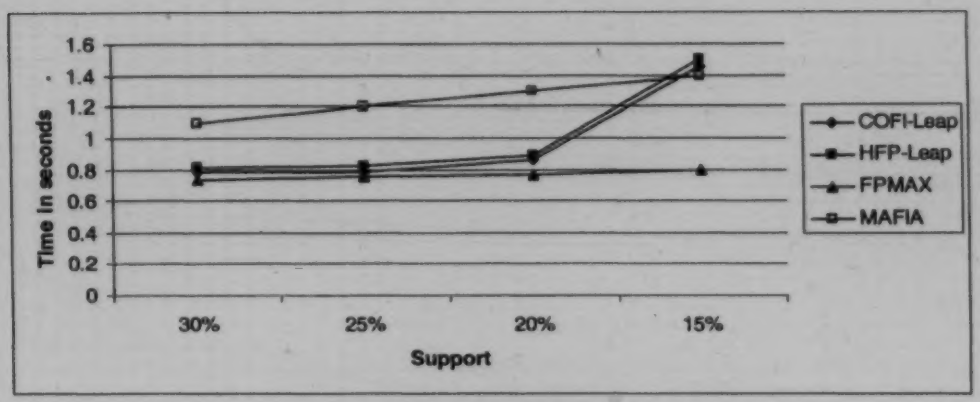


Figure 4.5: Mining Plant-Protein database

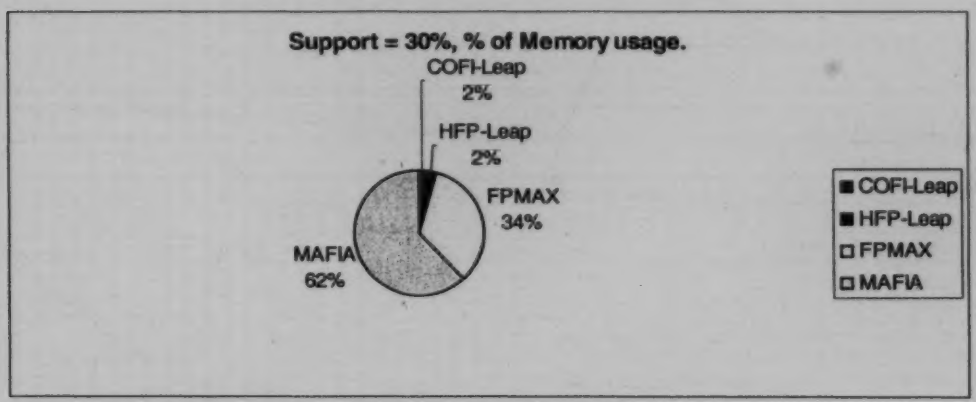


Figure 4.6: Memory usage while mining Plant-Protein database (support = 30%)

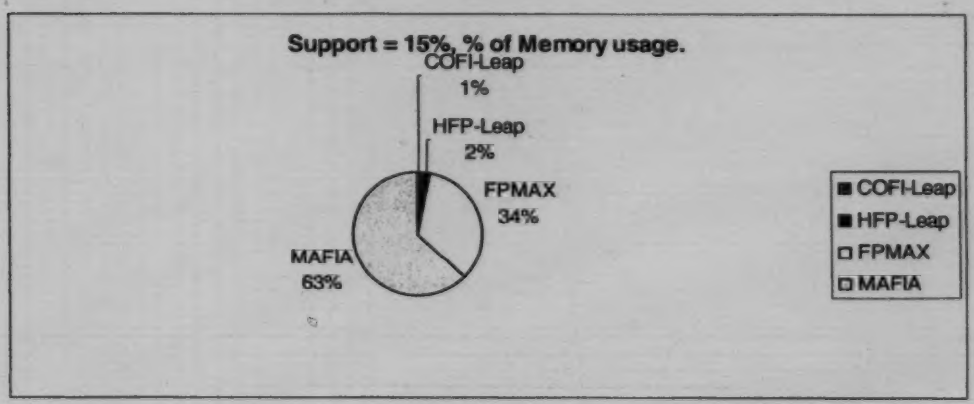


Figure 4.7: Memory usage while mining Plant-Protein database (support = 15%)

Table 4.1: Database characteristics

Databases	Number of items	Average transaction length	Number of transactions
Chess	76	37	3,196
Mushroom	120	23	8,124
Pumsb	7117	74	49,046
Accidents	572	45	340,184
T10K5DL12	5,000	12	10,000

Table 4.2: Mining different small datasets: The winner algorithms using high support threshold

Algorithms	All	CLOSED	MAXIMAL
Mushroom	FP-Growth	FP-CLOSED	FPMAX
Chess	COFI-ALL	COFI-CLOSED	COFI-MAX
Pumsb	FP-Growth	FP-CLOSED	GENMAX
Accidents	Eclat	CHARM	GENMAX
T10K5DL12	COFI-ALL	COFI-CLOSED	COFI-MAX

Table 4.3: Mining different small datasets: The winner algorithms using low support threshold

Algorithms	All	CLOSED	MAXIMAL
Mushroom	FP-Growth	FP-CLOSED	FPMAX
Chess	COFI-ALL	COFI-CLOSED	COFI-MAX
Pumsb	MAFIA	FP-CLOSED	COFI-MAX
Accidents	FP-Growth	FP-CLOSED	FPMAX
T10K5DL12	COFI-ALL	COFI-CLOSED	COFI-MAX

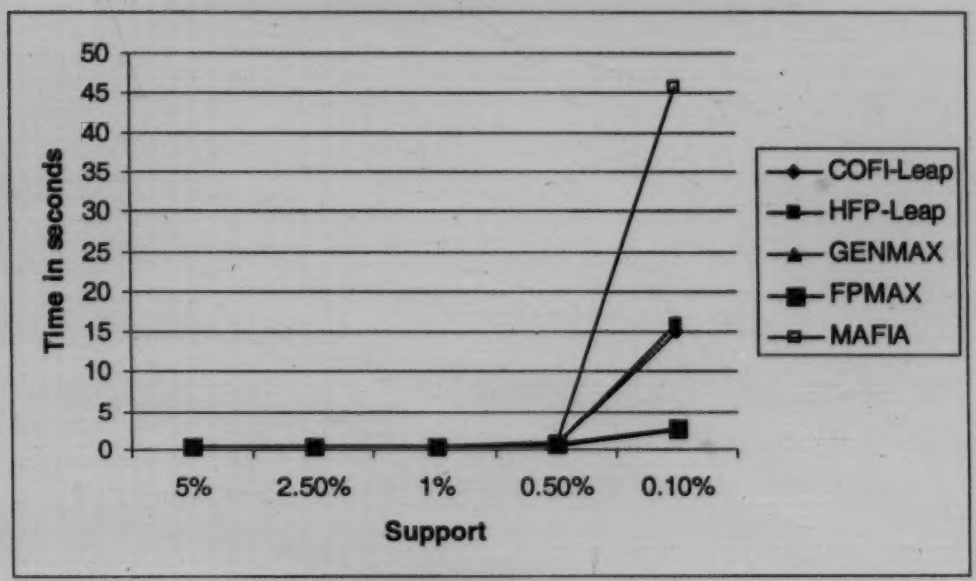


Figure 4.8: Mining retail database

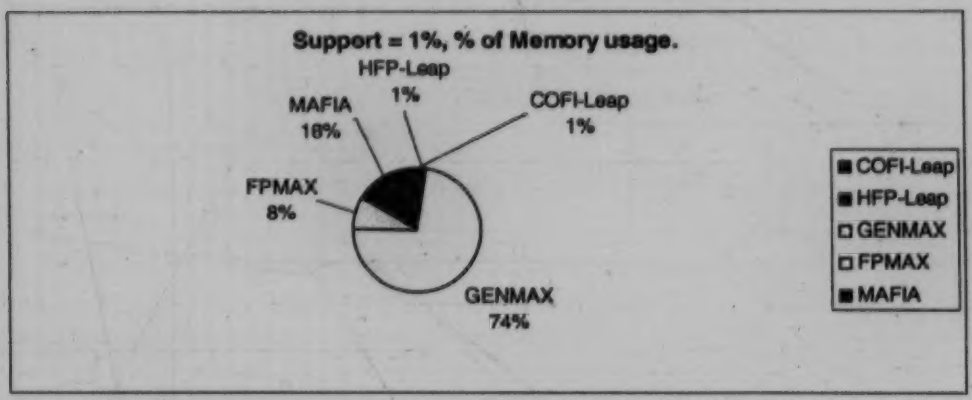


Figure 4.9: Memory usage while mining retail database (support = 1%)

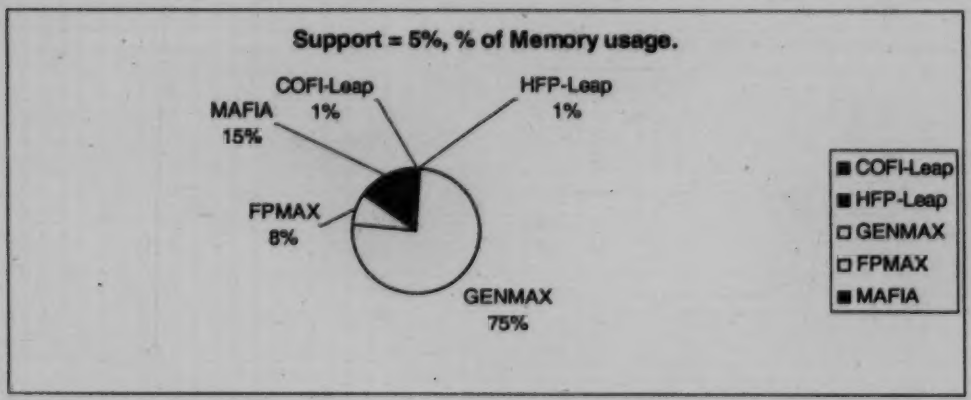


Figure 4.10: Memory usage while mining retail database (support = 5%)

experiments we confirmed the conclusion made at the FIMI 2003 workshop [43], that there are no clear winners with small databases. Indeed, algorithms that were shown to be winners with some databases were not the winners with others. Some algorithms quickly loose their lead once the support level becomes smaller. Tables 4.2 and 4.3, depicts as an example the winner algorithms for these datasets. In this table we see that almost every algorithm is a winner in at least one case. These experiments are depicted in Figures 4.11, 4.12, 4.13 and 4.14. All algorithms, except MAFIA, overlap in the figures. In these figures COFI-ALL, COFI-CLOSED, and COFI-MAX present Leap approach based on the COFI-trees idea.

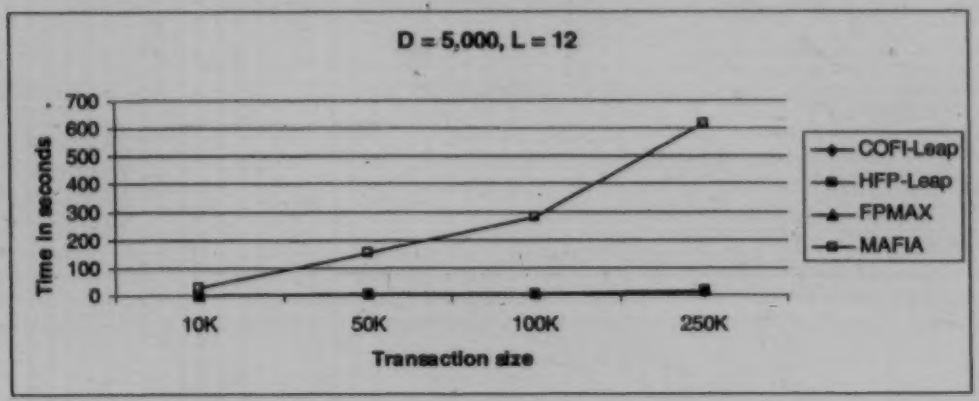


Figure 4.11: Mining synthetic database. D = 5000 items. Support = 0.5%

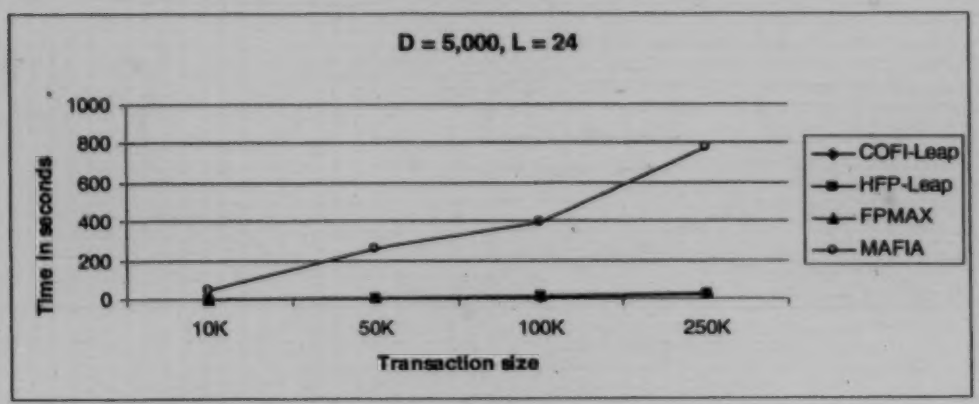


Figure 4.12: Mining synthetic database. D = 5000 items. Support = 0.5%

4.4.2 Mining Extremely large synthetic databases

To distinguish the subtle differences between Leap approach and FPMAX, we conducted our experiments on extremely large datasets. In this series of experiments, we used three synthetic datasets made of 5M, 25M, and 50M transactions, with a dimension equal to

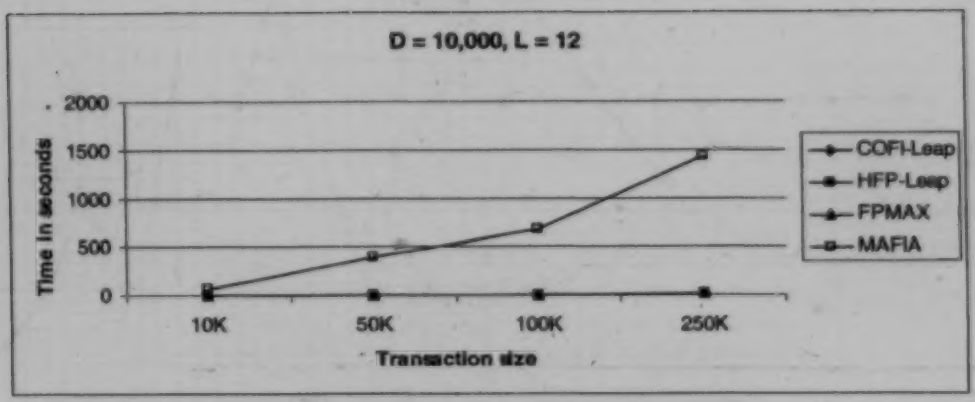


Figure 4.13: Mining synthetic database. D = 10000 items. Support = 0.5%

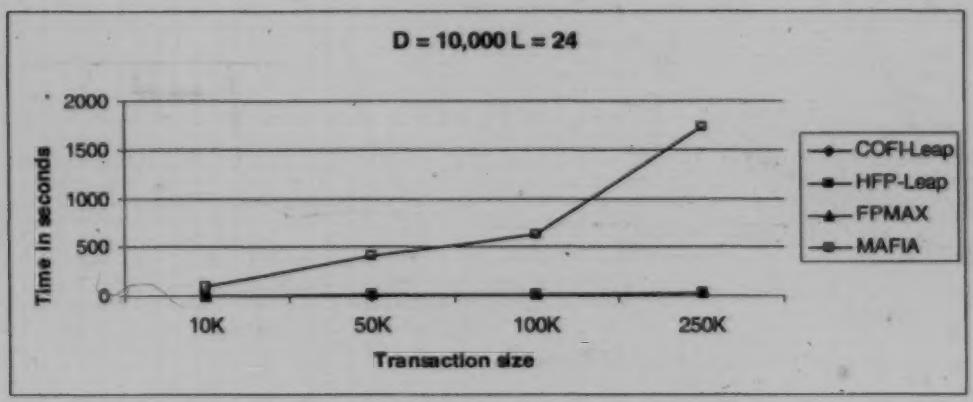


Figure 4.14: Mining synthetic database. D = 10000 items. Support = 0.5%

100K items, and an average transaction length equal to 24. All experiments were conducted using a support of 0.05%. In mining 5M transactions, the three algorithms show similar performances as COFI-Leap which finishes in almost less than 300 seconds. HFP-Leap finishes its work in 320 second while FPMAX finishes in 375 seconds. At 25M transactions, the difference starts to increase. The final test of mining a transactional database with 50M transactions, HFP-Leap discovers all patterns in 1980 second. COFI-Leap beats HFP-Leap by almost 100 seconds, while FPMAX finishes in 2985 seconds. The results, averaged on many runs, are depicted in Figure 4.15.

From these experiments, we see that the difference between FPMAX and Leap-based algorithms while mining synthetic datasets becomes clearer once we deal with extremely large datasets. Leap approaches save at least one third of the execution time compared to FPMAX. This is due to the reduction in candidate checking and to the lower memory requirements by the Leap-based approaches.

In the last set of experiments only FPMAX and Leap algorithms, that are COFI based, participated. *FP-Growth* was able to mine efficiently for all frequent patterns up to 5 millions. After that point, *FP-Growth* could not return a result within the wall time of 5000 seconds. COFI-ALL efficiently found the set of all patterns up to 100M transactions. For the set of closed itemsets and the set of maximal itemsets FP-CLOSED and FPMAX mined up to 50M transactions, while COFI-CLOSED and COFI-MAX mined all databases up to 100M transactions efficiently. All results are depicted in Figure 4.16. From these experiments we can see that the difference between FPMAX implementations and the Leap algorithms becomes clearer once we mine extremely large datasets. Leap saves at least one third of the execution time and in some cases goes up to half of the execution time compared to *FP-Growth* approach.

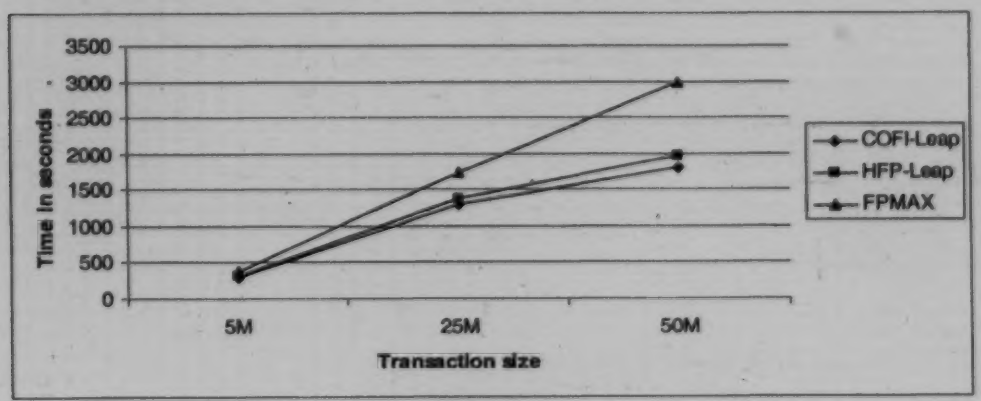


Figure 4.15: Mining extremely large database

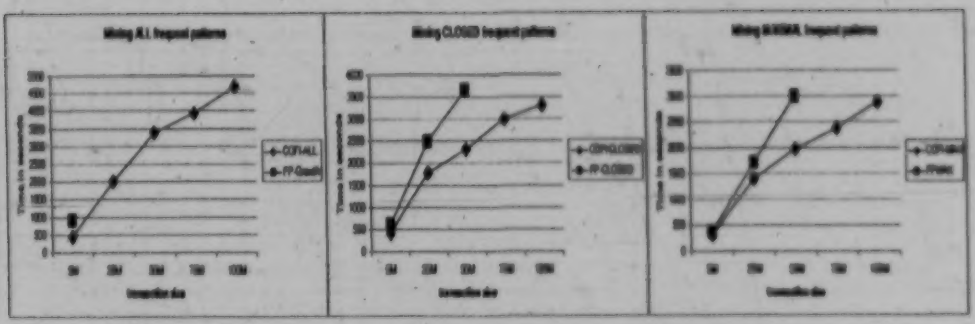


Figure 4.16: Scalability with very large datasets

4.4.3 Memory Usage

We also tested the memory usage by FPMAX, MAFIA HFP-Leap, and COFI-Leap while mining synthetic databases. In many cases we noticed that Leap algorithms consume one order of magnitude less memory than both FPMAX and MAFIA. Figure 4.17 illustrates a sample of the experiments that we conducted where the transaction size, the dimension and the average transaction length are respectively 1000K, 5K and 12. The support was varied from 0.1% to 0.01%.

This low memory usage observed by the Leap approach is due to the fact that HFP-Leap generates the maximal patterns directly from its HFP-Tree or from small chunks as in the case of COFI-trees. Also the intersection tree is never physically built. FPMAX, however, uses a recursive technique that keeps building trees for each frequent item tested and thus uses much more memory.

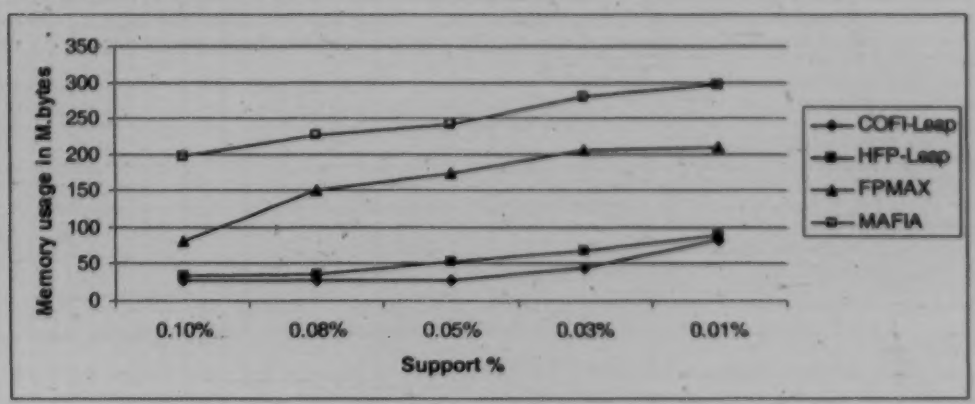


Figure 4.17: Memory usage

4.5 Summary

We presented a new way of traversing the pattern lattice to search for pattern candidates. The idea is to first discover maximal patterns and to keep enough intermediary information to generate from these maximal patterns all types of patterns with their exact support. Our new lattice traversal approach dramatically minimized the size of candidate list because it selectively jumps within the lattice toward the frequent pattern border. It also introduces a new method of counting the supports of candidates based on the supports of other candidate patterns, namely the branch supports of FPBs. Our performance studies show that our approach outperforms the state-of-the-art methods that have the same objective: discovering maximal and all patterns by, in some cases, two orders of magnitude, in particular, when mining for extremely large databases. This algorithm shows drastic saving in terms of memory usage as it has a small footprint in the main memory at any given time.

Chapter 5

Constraint-Based Mining with Leap-Traversal Approach

The condition that gives birth to a rule
is not the same as the condition to which the rule gives birth.
– Friedrich Nietzsche (1844 - 1900)

Mining for frequent itemsets can generate an overwhelming number of patterns, often exceeding the size of the original transactional database. One way to deal with this issue is to set filters and interestingness measures. Others advocate the use of constraints to apply to the patterns, either on the form of the patterns or on descriptors of the items in the patterns. However, typically the filtering of patterns based on these constraints is done as a post-processing phase. Filtering the patterns post-mining adds a significant overhead, suffering from the sheer size of the pattern set and losing the opportunity to exploit those constraints.

Ideally, dealing with the constraints should be done during the mining process as early as possible. In general, two types of constraints *monotone* and *anti-monotone* have been identified, and considering them early can significantly reduce the execution time. The idea has been exploited by some proposed approaches. However, most proposals consider either one of them but not both.

In this chapter we propose an approach that allows the efficient mining of frequent itemset patterns while pushing simultaneously both *monotone* and *anti-monotone* constraints during and at different strategic stages of the mining process.

5.1 Constraints Based Mining

Frequent Itemset Mining (FIM) is a key component of many algorithms that extract patterns from transactional databases. One challenge for frequent itemset mining is the potentially huge number of extracted patterns, which can eclipse the original database in size. In

addition to increasing the cost of mining, this makes it more difficult for users to find the valuable patterns. Introducing constraints to the mining process helps mitigate both issues. Decision makers can restrict discovered patterns according to specified rules. By applying these restrictions as early as possible, the cost of mining can be constrained. For example, users may be interested in purchases whose total price exceeds $100, or whose items cost between $50 and $100.

Constraint based mining is an ongoing area of research. Two important categories of constraints are *monotone* and *anti-monotone* [55]. *Anti-monotone* constraints are constraints that when valid for a pattern, are consequentially valid for any subset subsumed by the pattern. *Monotone* constraints when valid for a pattern are inevitably valid for any superset subsuming that pattern. The straightforward way to deal with constraints is to use them as a filter post-mining. However it is more efficient to consider the constraints during the mining process. This is what is referred to as "*pushing the constraints*" [67]. Most existing algorithms leverage (or push) one of these types during mining and postpone the other to a post-processing phase. New algorithms, such as Dualminer apply both types of constraints at the same time [15]. It considers these two types of constraints in a double process, one mirroring the other for each type of constraint, hence, its name. However, *monotone* and *anti-monotone* constraints do not necessarily apply in duality. Especially when considering the mining process as a set of distinct phases, such as the building of structures to compress the data and the mining of these structures, the application of these constraints differ by type. Moreover, some constraints have different properties and should be considered separately. For instance, minimum support and maximum support are intricately tied to the mining process itself while constraints on item characteristics, such as price, are not. There is no existing algorithm that efficiently pushes both types of constraints early in the mining process and neither traverses the lattice of patterns top-down nor bottom-up. We introduce herein an algorithm that pushes both *monotone* and *anti-monotone* constraints by wisely jumping several levels in the pattern lattice from the bottom and top, and cleverly reducing the unnecessary constraint checking while considering the intricacies and properties of the constraints and the patterns sought after.

The problem of discovering all frequent itemsets that satisfy constraints is a difficult one. The difficulty stems from the fact that, firstly, testing for minimum support and maximum support cannot be done simultaneously, since when valid, one is always true for subsets while the other is always true for supersets. Secondly, despite their selective power, some constraints cannot be checked to filter candidate itemsets until at a very late stage of the mining process.

We introduce a frequent itemset mining algorithm with the following properties:

- A *Leap-Traversal* strategy is used to apply constraints from selected nodes in the lattice, in contrast to bottom-up or top-down traversals.
- Both *monotone* and *anti-monotone* constraints are pushed efficiently by placing and timing their respective evaluation strategically.
- Regions where one constraint needs not be evaluated are identified quickly using proven theorems.
- Previously known data structures, such as FP-Tree [48] and COFI-tree, are used, but new algorithms exploiting these structures are proposed.
- Constraints are used not only to extract the valid frequent itemsets but also concurrently to obtain the valid frequent closed and maximal patterns along with their respective supports.

5.2 Constraints

As illustrated earlier in Section 2.5, a number of types of constraints have been identified in the literature [55]. In this work, we discuss two important categories of constraints *monotone*, and *anti-monotone*. A constraint ζ is *monotone* if and only if when an itemset X holds for ζ, then any superset of X also holds for ζ and a constraint ζ is *anti-monotone* if and only if when an itemset X violates ζ, then any superset of X also violates ζ.

5.2.1 Bi-directional Pushing of Constraints

Pushing constraints early means considering constraints while mining for patterns rather than postponing the checking of constraints until after the mining process. Given the intrinsic characteristics of existing algorithms for mining frequent itemsets, either going over the lattice of candidate itemsets top-down or bottom-up, considering all constraints while mining, is difficult. Most algorithms attempt to push either type of constraints during the mining process in hope of reducing the search space in one direction: from subsets to supersets or from supersets to subsets. Dualminer [15] pushes both types of constraints but at the expense of efficiency. Focusing solely on reducing the search space by pruning the lattice of itemsets is not necessarily a winning strategy. While pushing constraints early seems conceptually beneficial, in practice, the testing of the constraints can add significant overhead. If the constraints are not selective enough, checking the constraint predicates for each candidate can be onerous. It is thus important that we also reduce the checking frequency. While the primary benefit of early constraint checking is the elimination of candidates, which cannot pass the constraint, it can also be used to identify candidates which are guaranteed to pass the constraint and therefore do not need to be re-checked. In summary,

the goal of pushing constraints early is to reduce the itemset search space, eliminating unnecessary processing and memory consumption, while at the same time limiting the amount of constraint checking performed.

5.3 Leap with Constraints

The conjunction of all *anti-monotone* constraints comprises a predicate that we call $P()$. A second predicate $Q()$ contains the conjunction of the *monotone* constraints. A common approach is to include the ubiquitous minimum support constraint as part of $P()$. Similarly, the *monotone* maximum support constraint can be included as part of $Q()$. In this way, a frequent itemset mining algorithm can be extended to push $P()$ deeply by replacing checks for minimum support with checks for $P()$. In our algorithm, we separate the constraints on the support from other constraints. Thus, the minimum support constraint and the maximum support constraint are extracted from $P()$ and $Q()$ respectively. This is because checking for support is an integral part of the frequent itemset enumeration, while other constraints on item attributes are used for search space pruning.

Algorithm 10, *COFILeap* offers a number of opportunities to push the *monotone* and *anti-monotone* predicates $P()$ and $Q()$ respectively. We start this process by defining two terms which are head (H) and tail (T) where H is a *Frequent-Path-Base* or any subset generated from the intersection of *Frequent-Path-Bases*, and T is the itemset generated from intersecting all remaining *Frequent-Path-Bases* not used in the intersection of H. The intersection of H and T, $H \cap T$, is the smallest subset of H that may yet be considered. Thus, Leap focuses on finding frequent H that can be declared as local maximals and candidate global maximals. *BifoldLeap* (*COFILeap* with constraints) extends this idea to find local maximals that satisfy $P()$. We call these P-maximals.

Although we further constrain the P-maximals to itemsets that satisfy $Q()$, not all subsets of these P-maximals are guaranteed to satisfy $Q()$. To find the itemsets which satisfy both constraints, the subsets of each P-maximal are generated in order from long patterns to short ones. When a subset is found to fail $Q()$, further subsets do not need to be generated for that itemset, as they are guaranteed to fail $Q()$ also. There are three significant places where constraints can be pushed. These places are:

1. While building the FP-Tree.
2. While building the COFI-trees.
3. While intersecting the *Frequent-Path-Bases*, which is the main phase where both types of constraints are pushed at the same time (Algorithm 10).

Constraint pushing opportunities during FP-Tree construction.

There are two places where constraints can be pushed during the FP-Tree construction. These are:

1. Pushing $P()$ to each 1-itemset. Items that fail this test are not included in FP-Tree construction.

2. Pushing $Q()$ during the construction of FP-Tree paths. We use the idea from FP-Bonsai [11] where sub-transactions that do not satisfy $Q()$ are not used in the second phase of the FP-Tree building process. The supports for the items within these transactions are decremented. This may result in some previously frequent items becoming infrequent. Such items will not be used to construct COFI-trees in the following phase.

Constraint pushing opportunities during COFI-tree construction.

There are also two places where constraints can be pushed during the COFI-tree constructions. These are:

1. Q **Look ahead pushing:** Let X be the set of all items that will be used to build the COFI-tree, i.e. the items which satisfy $P()$ individually but have not been used as the root of a previous COFI-tree. If X fails $Q()$, there is no need to build the COFI-tree, as no subset of X can satisfy $Q()$.

2. P **Look ahead:** If X satisfies $P()$, there is also no need to test against $P()$ for this COFI-tree as all its subsets will also satisfy $P()$.

Constraint pushing opportunities during intersection of Frequent-Path-Bases.

In this phase, the actual leap is occurring and pushing constraints can happen in many cases. These are:

1. $P()$ and $Q()$ can be used to eliminate an itemset or remove the need to evaluate its intersections with additional *Frequent-Path-Bases*.

2. $P()$ and $Q()$ can be applied to the "head intersect tail" ($H \cap T$), which is the smallest subset of the current itemset that can be produced by further intersections.

These strategies are detailed in the following four theorems.

Theorem 1: If an intersection of *Frequent-Path-Bases* (H) fails $Q()$, it can be discarded, and there is no need to evaluate further intersections with H.

Proof: If an itemset fails $Q()$, all of its subsets are guaranteed to fail $Q()$ based on the definition of *monotone* constraints. Further intersecting H will produce subsets, all of which

are guaranteed to violate $Q()$.

Theorem 2: If an intersection of *Frequent-Path-Bases* (H) passes $P()$, it is a candidate P-maximal, and there is no need to evaluate further intersections with H.

Proof: Further intersecting H will produce subsets of H. By definition, no P-maximal is subsumed by another itemset which also satisfies $P()$. Therefore, none of these subsets of H are potential new P-maximals.

Theorem 3: If a node's $H \cap T$ fails $P()$, then the H node can be discarded, and there is no need to evaluate further intersections with H.

Proof: If an itemset fails $P()$, then all of its supersets will also violate $P()$ from the definition of *anti-monotone* constraints. Since a node's $H \cap T$ represents the subset of H that results from intersecting H with all remaining *Frequent-Path-Bases*, H and all combinations of intersections between H and remaining *Frequent-Path-Bases* are supersets of $H \cap T$ and therefore, guaranteed to fail $P()$ also.

Theorem 4: If a node's $H \cap T$ passes $Q()$, then $Q()$ is guaranteed to pass for any itemset resulting from the intersection of a subset of the *Frequent-Path-Bases* used to generate H plus the remaining *Frequent-Path-Bases* yet to be intersected with H. $Q()$ does not need to be checked in these cases.

Proof: $Q()$ is guaranteed to pass for all of these itemsets because they are generated from a subset of the intersections used to produce the $H \cap T$ and are therefore supersets of the $H \cap T$.

The following example, shown in Figure 5.1, illustrates how *BifoldLeap* works. A COFI-tree is made from five items, A, B, C, D, and E, with prices \$60, \$450, \$200, \$150, and \$100 respectively. In our example, this COFI-tree generates 5 *Frequent-Path-Bases*, $ACDE$, $ABCDE$, $ABCD$, $ABCE$, and $ABDE$, each with branch support one. The *anti-monotone* predicate, $P()$, is $Sum(Prices) \leq$ \$500, and the *monotone* predicate, $Q()$, is $Sum(prices) \geq$ \$100. Intersecting the first FPB with the second produces $ACDE$ which has a price of \$510, and therefore violates $P()$ and passes $Q()$. Next, we examine the $H \cap T$. The intersection of this node with the remaining three FPBs, yields itemset A with price \$60, passing $P()$ and failing $Q()$. None of these constraint checks provide an opportunity for pruning, so we continue intersecting this itemset with the remaining *Frequent-Path-Bases*. The first intersection is with the third FPB, producing ACD with price \$410, which satisfies

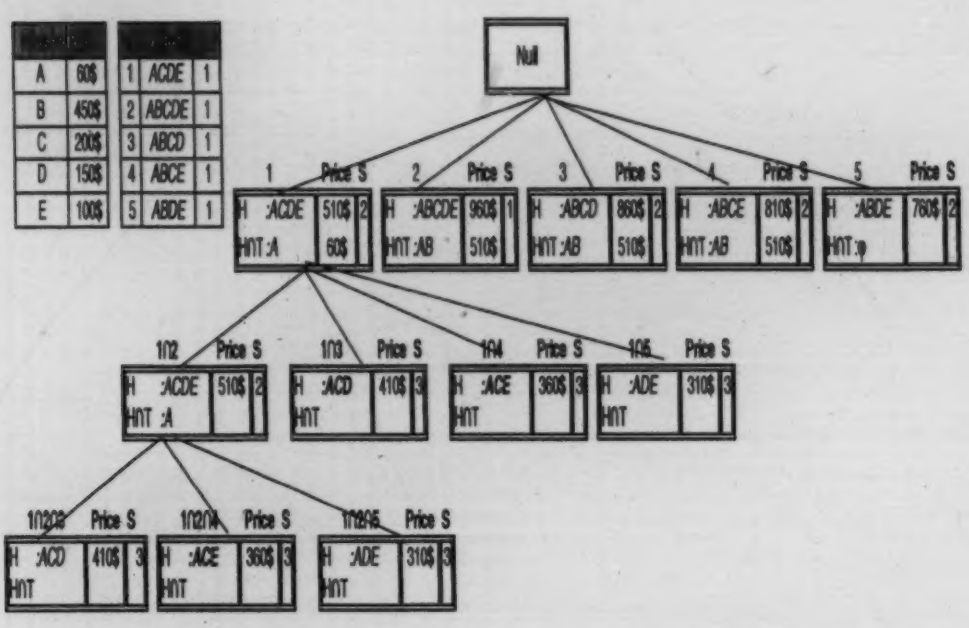


Figure 5.1: Pushing $P()$: $Sum(Prices) \leq \$500$ and $Q()$: $Sum(prices) \geq \$100$.

both the *anti-monotone* and *monotone* constraints. The second intersection produces ACE, which also satisfies both constraints. The same thing occurs with the last intersection, which produces ADE. Going back to the second *Frequent-Path-Base*, ABCDE, we find that the $H \cap T$, AB, violates the *anti-monotone* constraint with price \$510. Therefore, we do not need to consider $ABCDE$ or any further intersections with it. The remaining nodes are eliminated in the same manner. In total, three candidate P-maximals were discovered. We can generate all of their subsets while testing only against $Q()$. Finally, the support for these generated subsets can be computed from the existing *Frequent-Path-Bases*.

5.4 Performance Evaluation

To evaluate our *BifoldLeap* algorithm, we conducted a set of experiments to test the effect of pushing *monotone* and *anti-monotone* constraints separately, and then both in combination for the same datasets. To quantify scalability, we experimented with datasets of varying sizes. We also measured the impact of pushing versus post-processing constraints on the number of evaluations of $P()$ and $Q()$. Like in [15], we assigned prices to items using both uniform [38] and zipf [53] distributions. Our constraints consisted of conjunctions of tests for aggregate, minimum, and maximum price in relation to specific thresholds.

We received an FP-Bonsai code (based on *FP-Growth*) from its original authors [11]. Unfortunately, not all pruning and clever constraint considerations suggested in their FP-Bonsai paper were implemented in their code. Moreover, the implementation as received

Algorithm 10 *BifoldLeap*: Pushing P() and Q()

Input: D (transactional database); $\sigma; \Sigma$; *P(); Q()*.
Output: Frequent patterns satisfying P(), Q()
$PF1 \leftarrow$ Scan D to find the set of frequent P1-itemsets
$FPT \leftarrow$ Scan D to build FP-Tree using $PF1$ and $Q()$
$PGM(PGlobalMaximals) \leftarrow \emptyset$
for each item I in Header(FPT) **do**
  $LF \leftarrow FindlocalFrequentWithRespect(I)$
  Add $(I \cup LF)$ to PGM and Break IF $(P(I \cup LF))$
  break IF (Not $Q(I \cup LF)$)
  if Not $(I \cup LF) \subseteq PGM$ **then**
    $ICT \leftarrow$ Build COFI-tree for I
    $FPB \leftarrow$ FindFrequentPathBases(ICT)
    $PLM(PLMaximals) \leftarrow \{P(FPB)$ and frequent$\}$
    $InFrequentFPB \leftarrow notFrequent(FPB)$
    for each pair $(A, B) \in InFrequentFPB$ **do**
      header $\leftarrow A \cap B$
      Add *header* in PLM and Break IF (P(*header*) AND is frequent and not $\emptyset$)
      Delete *header* and break IF (Not Q(*header*))
      tail $\leftarrow$ Intersection(FPBs not in *header*)
      delete *header* and break IF (Not P(*header* $\cap$ *tail*))
      Do not check for $Q()$ in any subset of *header* IF (Q(*header* $\cap$ *tail*))
    end for
    for each pattern P in PLM **do**
      Add P in PGM IF ((P not subset of any $M \in PGM$)
    end for
  end if
end for
PQ-Patterns $\leftarrow$ GPatternsQ(FPB, PGM)
Output PQ-Patterns

produced some false positives and false negatives. This is why we opted not to add it to our comparison study.

We compared our algorithm with Dualminer [15]. Based on its authors' recommendations, we built the Dualminer framework on top of the MAFIA [16] implementation provided by its original authors.

Our experiments were conducted on a 3GHz Intel P4 with 2 GB of memory running Linux 2.4.25, Red Hat Linux release 9. The times reported also include the time used to output all matching itemsets. We tested these algorithms using both real datasets provided by Goethals and Zaki [43] and synthetic datasets generated using Quest [51]; we used 'retail' as our primary real dataset reported here. A dataset with the same characteristics as the one reported by Bucila et al. [15] was also generated.

5.4.1 Impact of $P()$ and $Q()$ selectivity on *BifoldLeap* and Dualminer

To differentiate between our novel *BifoldLeap* algorithm and Dualminer, we experimented against the retail dataset. In the first experiment (Figure 5.2), we pushed $P()$, then $Q()$, and finally $P() \wedge Q()$. We used the zipf distribution to assign prices to items. Both $P()$ and $Q()$ consisted of constraints on the sum of the prices. The constraint thresholds were chosen to not be very selective. Figure 5.3 presents the same experiment with more selective constraints, using *anti-monotone* and *monotone* constraints on the sum of the prices, and on the minimum and maximum item price. In this experiment, we found that *BifoldLeap* in most cases outperforms Dualminer and in some cases by more than one order of magnitude. The most interesting observation we found from this experiment was that if we push one type of constraint, e.g. $P()$, that takes $T1$ seconds and the other type of constraint, $Q()$, that takes $T2$ seconds where $T1 \leq T2$, in Dualminer pushing both constraints together will take $T3$ seconds, where $T3$ is always between $T1$ and $T2$. In contrast, *BifoldLeap* always takes less time with the conjunction of the constraints than with either constraint in isolation. *Monotone* and *anti-monotone* constraints can indeed mutually assist each other in the selectivity. *BifoldLeap* took better advantage of this reciprocal assistance in the pruning.

5.4.2 Scalability tests

Scalability is an important issue for frequent itemset mining algorithms. Synthetic datasets were generated with 50K, 100K, 250K, and 500K transactions, with 5K or 10K distinct items. In this experiment, *BifoldLeap* demonstrated extremely good scalability versus increasing dataset size. In contrast, Dualminer reached a point where it consumed almost three orders of magnitude more time than that needed by *BifoldLeap*. Figure 5.4.A depicts one of these results while mining datasets with only 5K unique items. As another experiment example, we tested both algorithms on datasets with up to 50 Million transactions and 100K items.

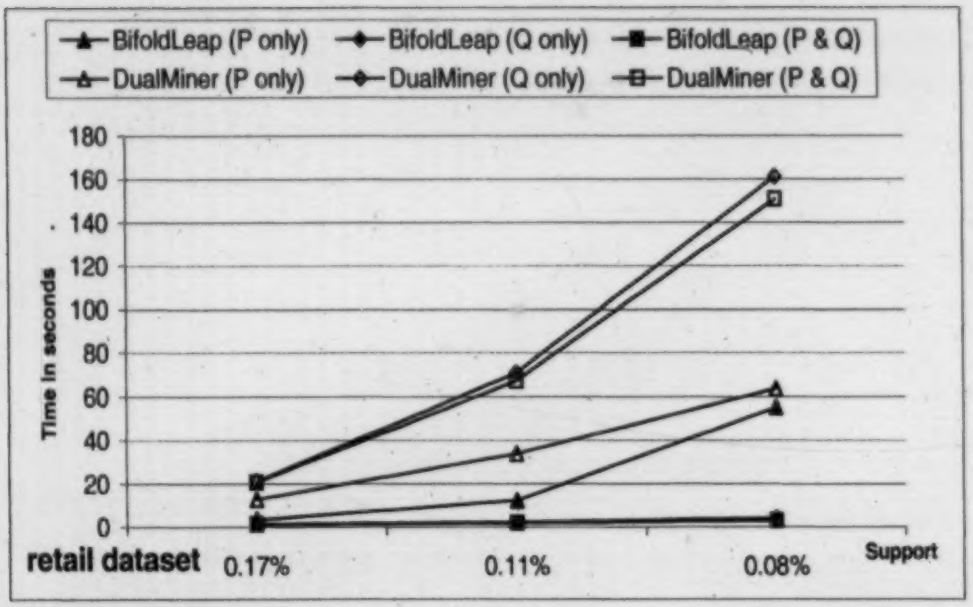


Figure 5.2: Pushing $P()$, $Q()$, and $P() \wedge Q()$. BifoldLeap $P() \wedge Q()$ is slightly better than BifoldLeap with $Q()$ only despite the fact that $Q()$ is the most selective. With DualMiner Combining $P() \wedge Q()$ is actually not helping.

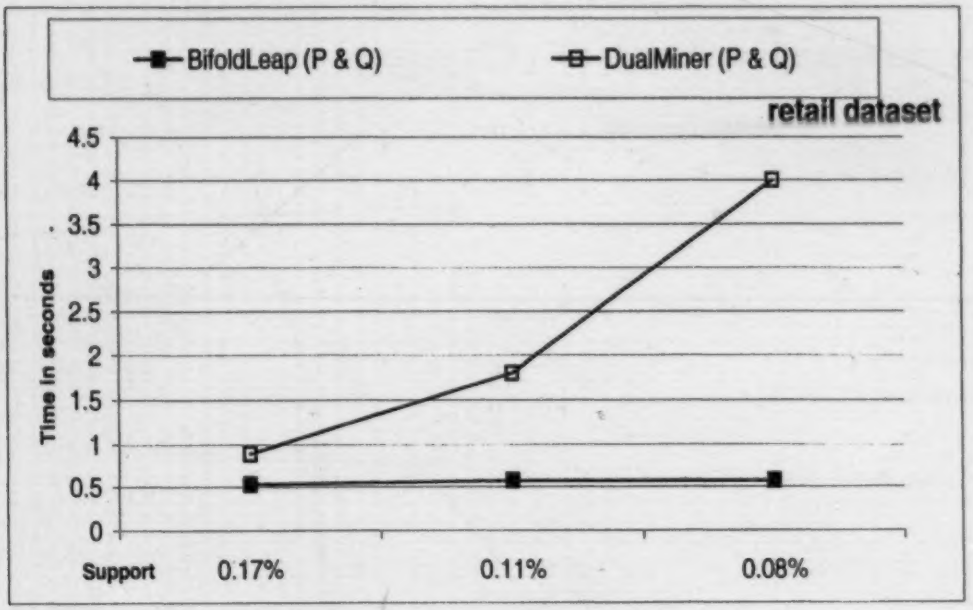


Figure 5.3: More selective constraints.

Dualminer finished the 1M dataset in 8534 seconds and failed to mine any larger datasets, while *BifoldLeap* finished in 186s, 190s, 987s and 2034s for the 1M, 5M, 25M and 50M transactions datasets respectively.

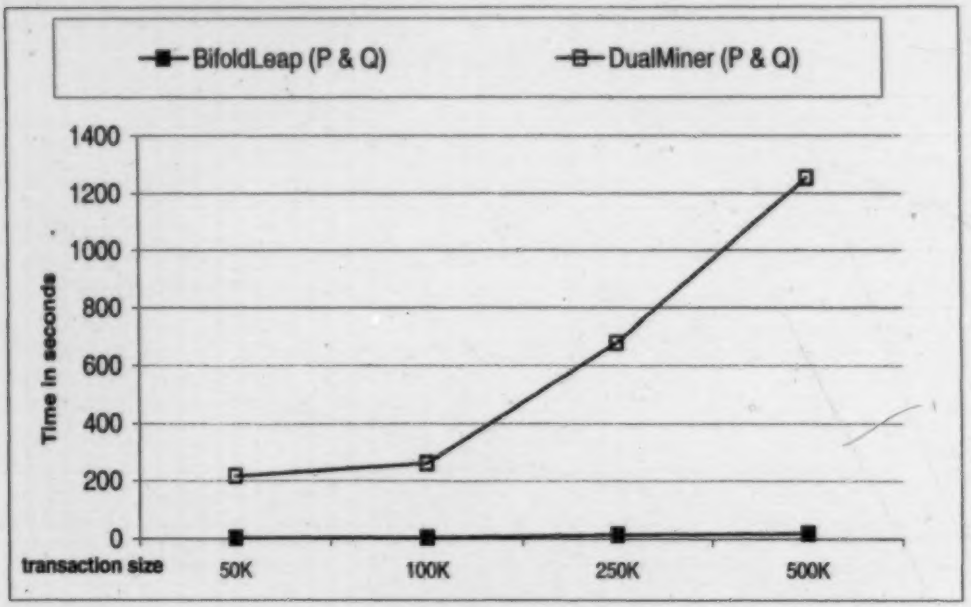


Figure 5.4: Scalability test

5.4.3 Constraint checking: pushing constraints versus post-processing

One of the major challenging issues for constraint mining is reducing the number of evaluations of $P()$ and $Q()$. In the following experiment, we generated a synthetic dataset with the same characteristics as the one reported by Bucila et al. [15]. Specifically, it was generated with 10,000 transactions, an average transaction length of 15, an average maximal pattern length of 10, 1000 unique items, and 10,000 patterns. We found that Dualminer was indeed good on this dataset as reported by Bucila et al. [15]. However, *BifoldLeap* outperformed it with the same order of magnitude as the tests on timing. This shows that the predicate checking is indeed a significant overhead and *BifoldLeap* outperforms Dualminer in time primarily because it does significantly less predicate checks.

The goal of these experiments was to test the number of evaluations and the effect of pushing constraints early versus post-processing them. We ran our experiments using this dataset with absolute support equal to 25, 50, and 75 using the two different distributions. We used a modified version of MAFIA with post-processing as the post-processing counterpart to Dualminer. Our implementation of Dualminer always tests minimum support and P() together, while *BifoldLeap*'s minimum support checks occur at different times and do

not contribute to the count for $P()$. Figure 5.5 depicts the results of these experiments. Our first observation is that Dualminer performs a huge number of constraint evaluations as compared to *BifoldLeap*. Even in cases where we only generated 255 patterns, Dualminer needed more than 50 thousand evaluations for both $P()$ and $Q()$, compared to almost 6 thousand needed by *BifoldLeap*. Our second observation is that MAFIA with post-processing requires fewer constraint evaluations than Dualminer itself.

Uniform-distribution

dmT10K1KD15L	Absolute support = 25		Absolute support = 50		Absolute support = 75	
	Early Push	Post Proc.	Early Push	Post Proc.	Early Push	Post Proc.
BifoldLeap (#P)	2266	3166	1483	1581	1212	1319
BifoldLeap (#Q)	4156	4650	299	351	166	189
DualMiner (#P)	25722	25649	18389	18028	14038	13720
DualMiner (#Q)	24946	298	17602	187	13221	160

# of generated patterns	255	158	134

(A)

Zipf-distribution

dmT10K1KD15L	Absolute support = 25		Absolute support = 50		Absolute support = 75	
	Early Push	Post Proc.	Early Push	Post Proc.	Early Push	Post Proc.
BifoldLeap (#P)	1814	2184	1428	1778	1207	1436
BifoldLeap (#Q)	420	625	125	312	99	254
DualMiner (#P)	11971	11722	8019	7790	6148	5950
DualMiner (#Q)	11185	197	7178	119	5282	100

# of generated patterns	130	76	62

(B)

Figure 5.5: Number of $P()$ and $Q()$ evaluations, using constraint pushing versus post-processing

5.4.4 Different distributions

All of our experiments were conducted on datasets using uniform and/or zipf price distributions. In most of the experiments, we found that the effect of changing the distribution on Dualminer was greater than for *BifoldLeap*. This can be justified by the effectiveness of the pruning techniques used by *BifoldLeap* that also reduce the number of candidate checks which consequently affected its performance. Figure 5.6 depicts one of these results for the retail dataset.

5.5 Summary

Since the introduction of association rules a decade ago and the launch of the research in efficient frequent itemset mining, the development of effective approaches for mining large transactional databases has been the focus of many research studies. Furthermore,

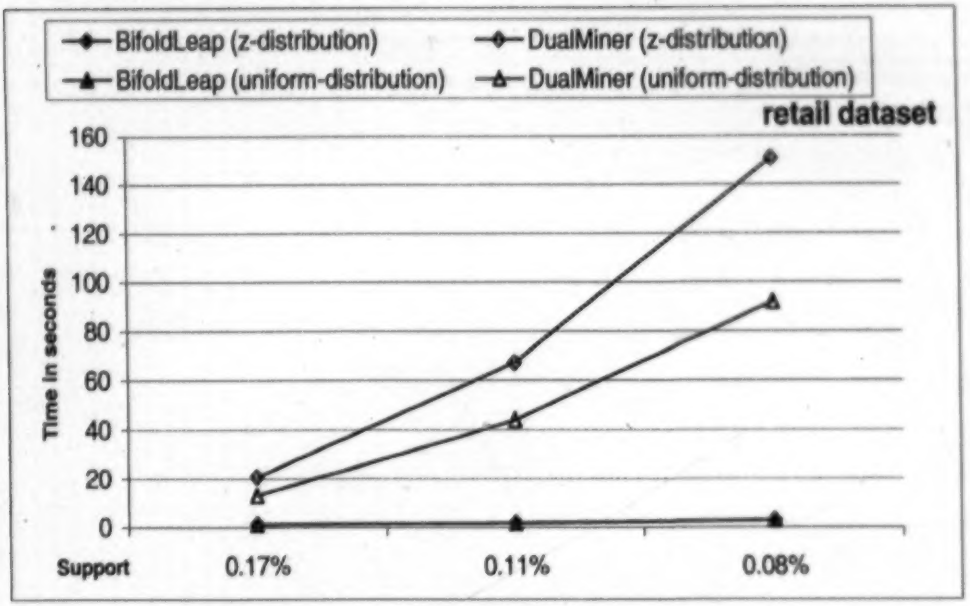


Figure 5.6: Effect of changing the price distribution

it is widely recognized that mining for frequent items or association rules, regardless of its efficiency, usually yields an overwhelming, crushing number of patterns. This is one of the reasons it is argued that the integration of data mining and database management technologies is required [17]. These large sets of discovered patterns could be queried. Expressing constraints using a query language could indeed help sift through the large pattern set to identify the useful ones. We argue that pushing the consideration of these constraints at the mining process before discovering the patterns is an efficient and effective way to solve the problem. This does not exclude the integration of data mining and database systems, but suggests the need for data mining query languages intricately integrated with the data mining process.

In this chapter we addresed the issue of early consideration of *monotone* and *anti-monotone* constraints in the case of frequent itemset mining. We proposed a Leap-Traversal approach, *BifoldLeap*, that traverses the search space by jumping from relevant node to relevant node and simultaneously checking for constraint violations. The approach we propose uses existing data structures, FP-Tree and COFI-tree, but introduces new pruning techniques to reduce the search costs. We conducted a battery of experiments to evaluate our constraint-based search. The experiments show the advantages of pushing both *monotone* and *anti-monotone* constraints as early as possible in the mining process despite the overhead of constraint checking. We also compared our algorithm to Dualminer, a state-of-the-art algorithm in constraint-based frequent itemset mining, and showed how our

algorithm outperforms it and can find frequent itemsets that satisfy constraints along with their exact supports.

Chapter 6

Parallel Leap-Traversal Approach

Nothing is more fairly distributed than common sense: no one thinks he needs more of it than he already has
– Descartes

When computationally feasible, mining extremely large databases produces tremendously large numbers of frequent patterns. In many cases, it is impractical to mine those datasets due to their sheer size; not only the extent of the existing patterns, but mainly the magnitude of the search space. As described before, many approaches have been suggested to sequentially mine for subsets of patterns such as maximal patterns or pushing constraints during the mining process. So far, those approaches are still not genuinely effective to mine extremely large datasets: database made up of multiples of millions of transactions or even billions of transactions. In this chapter, we propose a method that parallelizes each of the above mentioned strategies efficiently, i.e. mining in parallel for the set of maximal patterns and mining in parallel for the set of patterns while also pushing constraints. To the best of our knowledge, both strategies have never been proposed efficiently before. Using this approach we could mine significantly large datasets; with sizes never reported in the literature before. We are able to effectively discover frequent patterns in a database made of a billion transactions using a 32 processors cluster in less than 2 hours.

6.1 Headerless-Leap versus. COFI-Leap: What to parallelize?

In parallelizing the Leap approach the first question that comes to mind was *"which Leap strategy do we need to use (HFP-Leap or COFI-Leap)?"*; especially considering that both strategies show good performance results mainly when mining for extremely large databases. To answer the above question we studied the characteristics of each method.

HFP-Leap has the following characteristics:

- It generates one HFP-Tree for the whole database. Consequently, it requires one tree of intersections to generate the full set of maximal patterns, or the set of patterns that satisfies the constraints.
- When the pruning algorithms are applied on the tree, huge chunks of the intersections can be avoided.

While the COFI-Leap has the following advantages

- It is an iterative process where small independent trees are built that do not consume lots of memory.
- Pruning techniques can be applied at the COFI-tree constructions phase, even before the Leap approach is started.

When we decide to choose a method to parallelize we found that the parallel version of Leap favours using leap intersections on the HFP-Leap over the COFI-Leap approach, in spite of the fact that, COFI-Leap seems more fit for a parallel implementation intuitively.

The reason for this is that COFI trees are built on FP-Tree. Parallel implementations of COFI distributed FP-Trees generate local COFI-trees. To broadcast these COFI-trees we need to send at least one message per COFI-tree, i.e. if we have n COFI-trees then n messages will be broadcast. However, for HFP-Leap we apply one set of intersections per tree, i.e. per node. As a result of this, only one set of messages is needed per node to broadcast the set of local maximals among processors. Reducing the number of messages is the main motivation for choosing the HFP-Leap over COFI-Leap.

The work in this chapter is divided into two main parts: the Parallel-Leap that generates the set of Maximal patterns in parallel with the set of *Frequent-Path-Bases* to generate all patterns if needed; and the parallel-*BifoldLeap*, i.e., mining for constraint frequent pattern in parallel.

6.2 Parallel-Leap-Traversal Approach

The parallel-Leap-Traversal approach starts by partitioning the data among the parallel nodes, where each processor receives almost the same number of transactions. In our experiments we simply generated the transactions for each processors separately and independently. Each processor scans its partition to find the frequency of candidate items. The list of all supports is reduced to the master node to get the global list of frequent 1-itemsets.

The second scan of each partition starts with the goal of building a local Headerless Frequent Patterns tree. The local set of *Frequent-Path-Bases* is generated from each tree.

Those sets are broadcast to all processors. Identical *Frequent-Path-Bases* are merged and sorted lexicographically, the same as with the sequential process. At this stage the *Frequent-Path-Bases* are split among the processors. Each processor is allocated a carefully selected set of *Frequent-Path-Bases* to build their respective intersection trees. This distribution is discussed further below. Pruning algorithms are applied at each processor to reduce the size of the intersection trees [83]. Local maximal patterns are generated at each node. Each processor then sends its maximal patterns to one master node, which filters them to generate the set of global maximal patterns. Algorithm 11 presents the steps needed to generate the set of maximal patterns in parallel. Figure 6.1 presents the steps needed by Algorithm 11 to generate the frequent patterns.

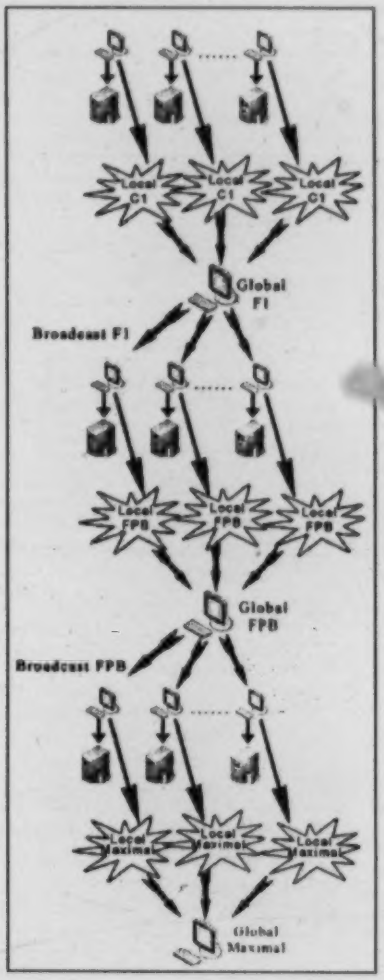


Figure 6.1: Parallel Leap-Traversal Steps

6.2.1 Load Sharing among Processors

While the trees of intersections are not physically built, they are virtually traversed to complete the relevant intersections of *Frequent-Path-Bases*. Since each processor can handle independently some of these trees, and the sizes of these trees of intersections are monotonically decreasing, it is important to cleverly distribute these among the processors to avoid significant load imbalance. A naïve and direct approach would be to divide the trees sequentially. Given p processors we would give the first $\frac{1}{p}^{th}$ trees to the first processor, the next fraction to the second processor, and so on. This strategy unfortunately leads to eventual imbalance between processors since the last processor getting all small trees would undoubtedly terminate before other nodes in the cluster. A more elegant and effective approach would be a round robin approach, considering the sizes of the trees. When ordered by size, the first p trees are distributed one to each processor and so on for each set of p trees. This avoids having a processor dealing with only large trees while another processor is intersecting with only small ones. Again this strategy may still create imbalance between processors, however, less acutely than the naïve direct approach. The strategy that we propose, and call First-Last, distributes two trees per processor at a time. The largest tree and the smallest tree are assigned to the first processor, then the second largest tree and penultimate small tree to the second processor, the third largest tree and third smallest tree to the third processor and so on in a loop. This approach seems to advocate a better load balance as was demonstrated by our experiments.

Algorithm 11 Parallel-HFP-Leap: Parallel-Leap-Traversal with Headerless FP-Tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns with their respective supports.

- D is already distributed. If not then partition D among the available p processors;
- Each processor p scans its local partition D_p to find the set of local candidate 1-itemsets L_pC1 with their respective local support;
- The supports of all L_iC1 are transmitted to the master processor;
- Global Support is counted by master and $F1$ is generated;
- $F1$ is broadcast to all nodes;
- Each processor p scans its local partition D_p to build the local Headerless FP-Tree L_pHFP based on $F1$;
- $L_pFPB \leftarrow$ FindFrequentPatternBases(L_pHFP);
- All L_pFPB are sent to the master node ;
- Master node generates the global FPB from all L_pFPB;
- The global FPB are broadcast to all nodes;
- Each processor p is assigned a set of local header nodes LHD from the global FPB; {this is the distribution of trees of intersections}

for each i in LHD for each processor **do**
  $LOCALMaximals \leftarrow$ FindMaximals(FPB, σ);
end for

- Send all $LOCALMaximals$ to the master node;
- The master node prunes all $LOCALMaximals$ that have supersets itemsets in $LOCALMaximals$ to produce $GLOBALMaximals$;
- The master node outputs $GLOBALMaximals$.

6.2.2 Parallel-Leap-Traversal Approach : An Example

The following example illustrates how the Leap-Traversal approach is applied in parallel. Figure 6.2.A presents 7 transactions made of 8 distinct items which are: A, B, C, D, E, F, G, and H. Assuming we want to mine those transactions with a support threshold equal to at least 3, using two processors, Figure 6.2 illustrates all the needed steps to accomplish this task. The database is partitioned among the two processors where the first three transactions are assigned to the first processor, $P1$, and the remaining ones are assigned to the second processor, $P2$ (Figure 6.2.A).

In the first scan of the database, each processor finds the local support for each item: $P1$ finds the support of A, B, C, D, E, F and G which are 3, 2, 2, 2, 2, 1 and 2 respectively, and $P2$ the supports of A, B, C, D, E, F, and H which are 2, 3, 3, 3, 3, 3, 2. A reduce operation is executed to find that the global support of A, B, C, D, E, F, G, and H items is 5, 5, 5, 5, 5, 4, 2, and 2. The last two items are pruned as they do not meet the threshold criteria (support > 2), and the remaining ones are declared frequent items of size 1. The set of Global frequent 1-itemset is broadcast to all processors using the first round of messages.

The second scan of the database starts by building the local headerless tree for each processor. From each tree, the local *Frequent-Path-Bases* are generated. In $P1$ the *Frequent-Path-Bases* $ABCDE$, ABE, and $ACDF$ with branch support equal to 1 are generated. $P2$ generates $ACDEF$, $BCDF$, BEF, and $ABCDE$ with branch supports equal to 1 for all of them (Figure 6.2.B). The second set of messages is executed to send the locally generated *Frequent-Path-Bases* to $P1$. Here, identical ones are merged and the final global set of *Frequent-Path-Bases* are broadcast to all processors with their branch support (Figure 6.2.C).

Each processor is assigned a set of header nodes to build their intersection tree as in Figure 6.2.D. In our example, the first, third, and sixth *Frequent-Path-Bases* are assigned to $P1$ as header nodes for its intersection trees. $P2$ is assigned to the second, fourth, and fifth *Frequent-Path-Bases*. The first tree of intersection in $P1$ produces $ACDE$, BCD, and ABE, with support equal to 3, 3, and 3 respectively. The second assigned tree produces CDF with support equal to 3. $P1$ produces 4 local maximals which are BE, AE, BE, CDF with support equal to 4, 4, 4, 3 respectively. $P2$ produced CDF, BE, and AE with support equal to 3, 4, and 4 respectively. All local maximals are sent to $P1$ in which any local maximal that has any other superset of local maximals from other processors are removed. The remaining patterns are declared as global maximals (Figure 6.2.E).

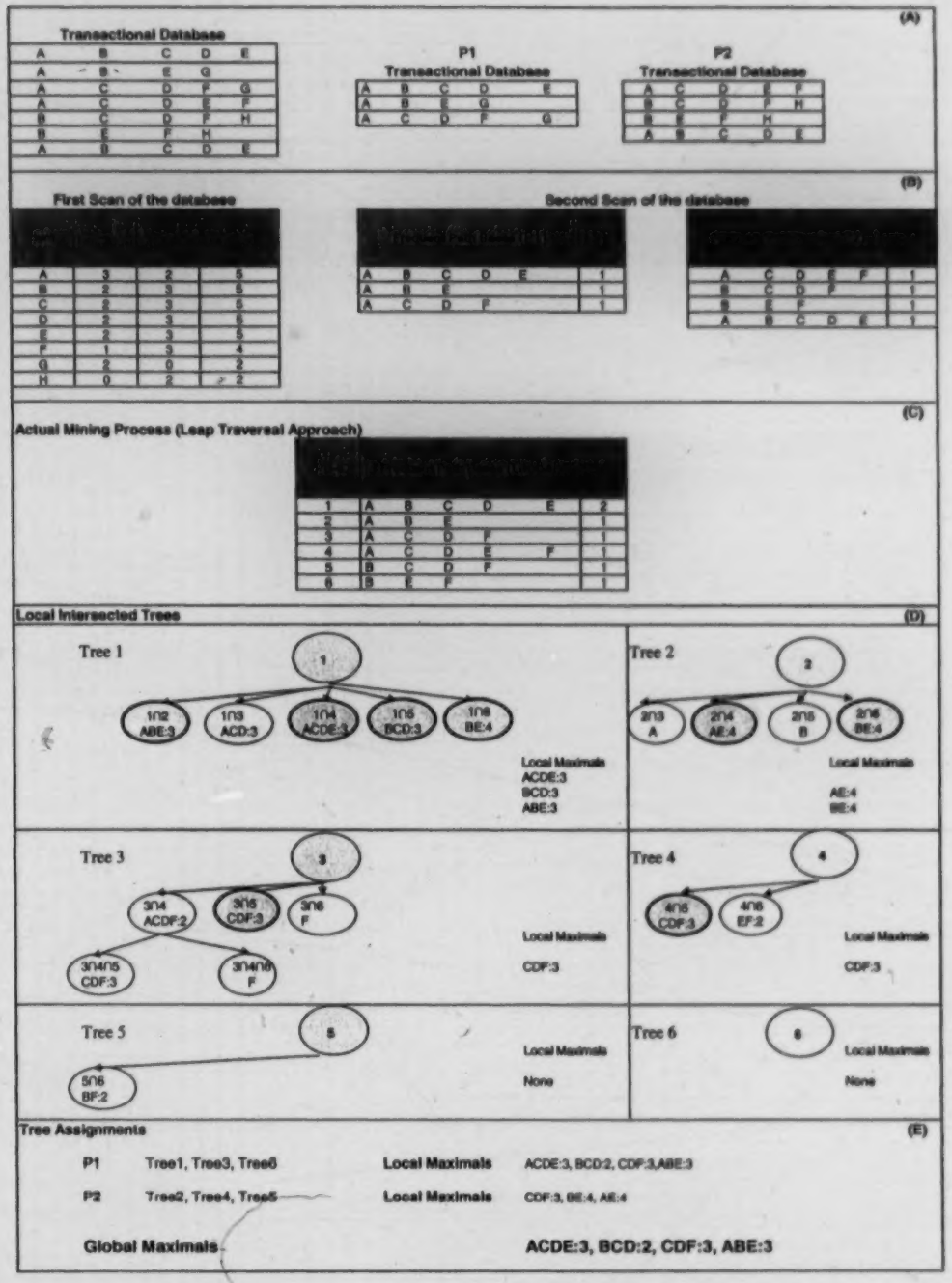


Figure 6.2: Example of Parallel-Leap-Traversal: Finding and intersecting the *Frequent-Path-Bases*

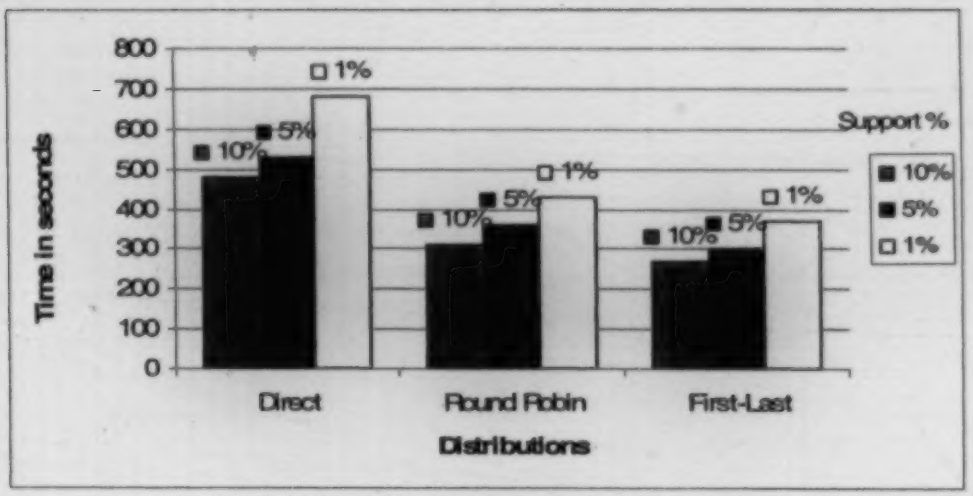


Figure 6.3: Effect of Leap node distributions

6.3 Performance Evaluations

To evaluate our parallel Leap-Traversal approach, we conducted a set of different experiments using a cluster made of twenty boxes. Each box has Linux 2.4.18, dual processor 1.533 GHz AMD Athlon MP 1800+, 1.5 GB of RAM. Nodes are connected by Fast Ethernet and Myrinet 2000 networks. In this set of experiments, we generated synthetic datasets using [51]. All transactions are made of 100,000 distinct items with an average transaction length of 12 items per transaction. The size of the transactional databases used varies from 100 million transactions to 1 billion transactions.

With our best efforts and literature searches, we were unable to find a parallel frequent mining algorithm that could mine more than 12 million transactions. The *Asynchronous Parallel Mining* algorithm claims this results achievement [19]. A parallelization of *FP-Growth* by Pramudiono and Kitsuregawa [69] mines transaction sets that do not exceed 100,000 transactions. The parallel implementation of *MaxMiner* by Cheung and Luo [21] mined only 2 million transactions. This is far less than our target size environment. Due to this large discrepancy in transaction capacity, we did not compare our algorithm against any other existing algorithms.

We conducted a battery of tests to evaluate the processing load distribution strategy, the scalability vis-à-vis the size of the data to mine, and the speed-up gained from adding more parallel processing power. Some of the results are portrayed hereafter.

6.3.1 Effect of Load Distribution Strategy

We enumerated above three possible strategies for tree of intersection distribution among the processors. As explained, the trees are in decreasing order of size and they can either be distributed arbitrarily using the naïve approach, or more evenly using a round robin approach, or finally with our suggested First-Last approach.

The naïve and simple strategy uses a direct and straightforward distribution. For example, if we have 6 trees to assign to 3 processors, the first two trees are assigned to the first processor, the third and fourth trees are assigned to the second processor, and the last two trees are assigned to the last processor. Knowing that the last trees are smaller in size than the first trees, the third processor will inevitably finish before the first processor. In the round robin distribution, the first, second and third tree are allocated respectively to the first, second and third processor and the remaining forth, fifth and sixth trees are assigned respectively to processor one, two and three. With the last strategy of distribution, First-Last, the trees are assigned in pairs: processor one works on the first and last tree, processor two receives the second and fifth tree, while the third processor obtains the third and fourth trees.

From our experiments in Figure 6.3 using 100 million transaction, we can see that the First-Last distribution gave the best results. This can be justified by the fact that since trees are lexicographically ordered then in general trees on the left are larger than those on the right. By applying the First-Last distributions we always try to assign largest and smallest tree to the same node. All our remaining experiments use the First-Last distribution methods among intersected trees.

6.3.2 Scalability with respect to the Database Size

One of the main goals in this work is to mine extremely large datasets. In this set of experiments we tested the effect of mining different databases made of different transactional databases varying from 100 million transactions up to one billion transactions. To the best of our knowledge, experiments with such big sizes have never been reported in the literature. We mined those datasets using 32 processors, with three different support thresholds: 10%, 5% and 1%. We were able to mine one billion transactions in 5020 seconds for a support of 10% up to 6100 seconds for a support of 1%. Figure 6.4 shows the results of this set of experiments. While the curve does not illustrate a perfect linearity in the scalability, the execution time for the colossal one billion transaction dataset was a very reasonable one hour and forty one minutes with a 1% support and 32 relatively inexpensive processors.

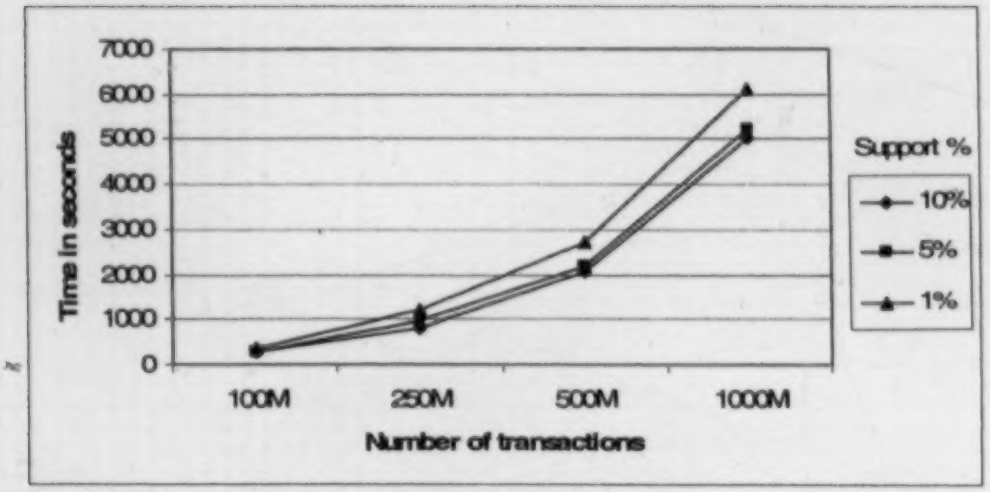


Figure 6.4: Scalability with respect to the transaction size, (number of processors = 32)

6.3.3 Scalability with respect to the Number of Processors

To test the speed-up of our algorithm with the increase number of of processors, we fixed the size of the database to 100 million transactions and examined the execution time on this dataset with one processor up to 32 processors. The execution time is reduced sharply when two to four parallel processors are added; it continues to decrease significantly afterward with additional processors (Figure 6.5). The speedup was fairly acceptable as almost two folds were achieved with 4 processors, 4 folds while using 8 processors, and almost 13 folds while using 32 processors. These results are depicted in Figure 6.6. As for Figure 6.5, we noticed that for the support of 1% with 32 processors it took 435 seconds to mine the 100 million transactions. Given the same approximate time, a sequential algorithm in one processor would mine only 5 million transactions. (FP-MAX in 406 seconds, HFP-Leap in 404 seconds, and COFI-Leap in 394 seconds). In the figure, one processor using the sequential algorithm mined the 100 million transactions in 5400 seconds. Given the same time, with 32 processors we could mine almost 1 billion transactions (6100 seconds for one billion).

6.4 Parallel *BifoldLeap*

The parallel *BifoldLeap* starts by partitioning the data among the parallel nodes using the same method in Section 6.2, where each node receives almost the same number of transactions. Each processor scans its partition to find the frequency of candidate items. The list of all supports is reduced to the master node to get the global list of frequent 1-itemsets. The second scan of each partition starts with the goal of building a local headerless

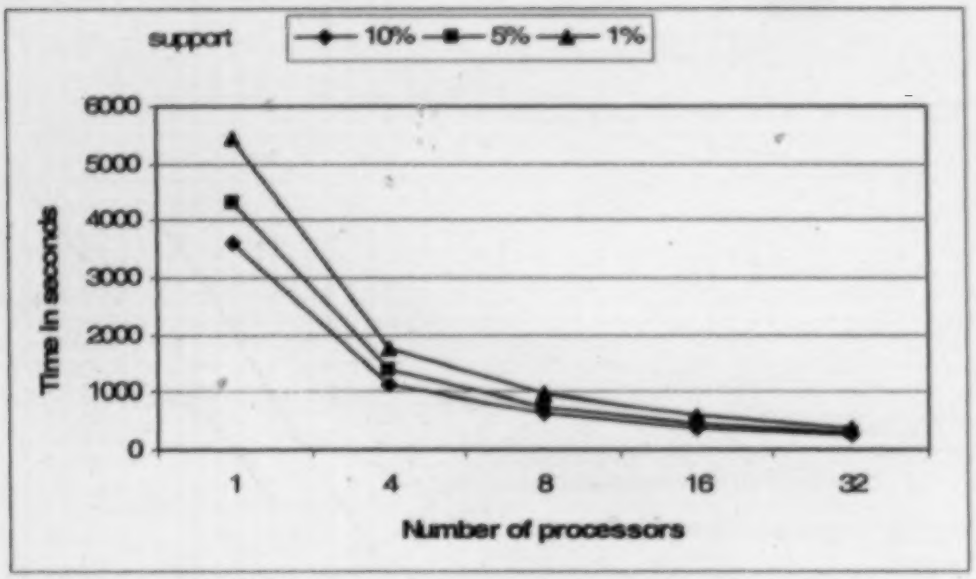


Figure 6.5: Scalability with respect to the number of processors, (transaction size = 100M)

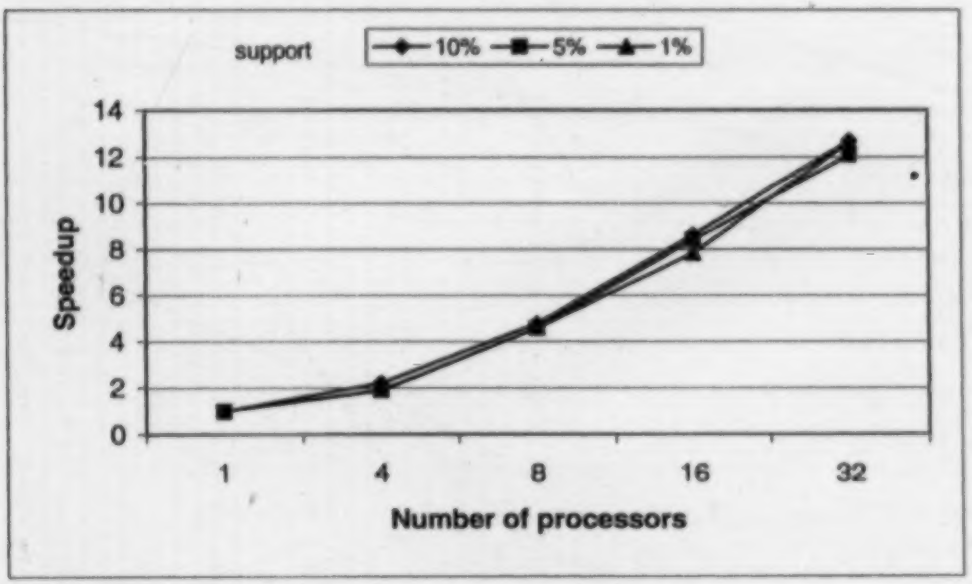


Figure 6.6: Speedup with different support values, (transaction size = 100M)

frequent patterns tree. From each tree, the local set of *Frequent-Path-Bases* is generated. Those sets are broadcast to all processors. Identical *Frequent-Path-Bases* are merged and sorted lexicographically, the same as with the sequential process. At this stage the *Frequent-Path-Bases* are split among the processors. Each processor is allocated a carefully selected set of *Frequent-Path-Bases* to build their respective intersection trees, with the goal of creating similar depth trees among the processors. Pruning algorithms are applied at each processor to reduce the size of the intersection trees as it is done in the sequential version [83]. Maximal patterns that satisfy the $P()$ constraints are generated at each node. Each processor then sends its P-maximal patterns to one master node, which filters them to generate the set of global P-maximal patterns and then finds all their subsets that satisfy $Q()$. Algorithm 12 presents the steps needed to generate the set of patterns satisfying both $P()$ and $Q()$ in parallel.

Algorithm 12 Parallel-HFP-*BifoldLeap*: Parallel-*BifoldLeap* with Headerless FP-Tree

Input: D (transactional database); $P()$; $Q()$; and σ (Support threshold).
Output: Patterns satisfying both $P()$ and $Q()$ with their respective supports.
- D is already distributed otherwise partition D among the available p processors;
- Each processor p scans its local partition D_p to find the set of local candidate 1-itemsets L_pC1 with their respective local support;
- The supports of all L_iC1 are transmitted to the master processor;
- Global Support is counted by master and $F1$ is generated;
- $F1$ is broadcast to all nodes;
- Each processor p scans its local partition D_p to build the local Headerless FP-Tree L_pHFP based on $F1$;
- $L_pFPB \leftarrow$ FindFrequentPatternBases(L_pHFP);
- All L_pFPB are sent to the master node ;
- Master node generates the global FPB from all L_pFPB;
- The global FPB are broadcast to all nodes;
- Each processor p is assigned a set of local header nodes LHD from the global FPB; {this is the distribution of trees of intersections}

for each i in LHD in each processor **do**
  $LOCAL-P-Maximals \leftarrow$ Find-P-Maximals($FPB, \sigma, P(), Q()$);
end for
- Send all $LOCAL-P-Maximals$ to the master node;
- The master node prunes all $LOCAL-P-Maximals$ that have supersets itemsets in $LOCAL-P-Maximals$ to produce $GLOBAL-P-Maximals$;
- The master node generates frequent patterns satisfying both P() and Q() from $GLOBAL-P-Maximals$.

6.4.1 Parallel *BifoldLeap*: An example

The following example illustrates how the *BifoldLeap* approach is applied in parallel. Figure 6.7.A presents 7 transactions made of 8 distinct items which are: A, B, C, D, E, F, G, and H with prices \$10, \$20, \$30, \$40, \$50, \$60, and \$70 respectively. Assuming we want to mine those transactions with a support threshold equal to at least 3 and to generate patterns where their total prices are between \$30 and \$100 (i.e $P()$: $Sum\ Of\ Prices <$ \$100, and $Q()$: $Sum\ Of\ Prices >$ \$30, using two processors. Figures 6.7.A and 6.7.B illustrate all the needed steps to accomplish this task. The database is partitioned among the two processors where the first three transactions are assigned to the first processor, $P1$, and the remaining

ones are assigned to the second processor, $P2$ (Figure 6.7.A).

In the first scan of the database, each processor finds the local support for each item: $P1$ finds the support of A, B, C, D, E, F and G which are 3, 2, 2, 2, 2, 1 and 2 respectively, and $P2$ the supports of A, B, C, D, E, F, and H which are 2, 3, 3, 3, 3, 3, 2. A reduced operation is executed to find that the global support of A, B, C, D, E, F, G, and H items is 5, 5, 5, 5, 5, 4, 2, and 2. The last two items are pruned as they do not meet the threshold criteria (support > 2), and the remaining ones are declared frequent items of size 1. The set of global frequent 1-itemsets is broadcast to all processors using the first round of messages.

The second scan of the database starts by building the local headerless FP-Tree for each processor. From each tree, the local *Frequent-Path-Bases* are generated. In $P1$ the *Frequent-Path-Bases* $ABCDE$, ABE, and $ACDF$ with branch support equal to 1 are generated. $P2$ generates $ACDEF$, $BCDF$, BEF, and $ABCDE$ with branch supports equal to 1 for all of them (Figure 6.7.B). The second set of messages is executed to send the locally generated *Frequent-Path-Bases* to $P1$. Here, identical ones are merged and the final global set of *Frequent-Path-Bases* are broadcast to all processors with their branch support. (6.7.C)

Each processor is assigned a set of header nodes to build their intersection tree as in Figure 6.8.D. In our example, the first, third, and sixth *Frequent-Path-Bases* are assigned to $P1$ as header nodes for its intersection trees. $P2$ is assigned to the second, fourth, and fifth *Frequent-Path-Bases*. The first tree of intersection in $P1$ produces 3 P-maximals (i.e. with total prices is less than \$100) BCD : \$90, ABE : \$80, and ACD : \$80 with support equal to 3, 3, and 4 respectively. The second assigned tree does not produce any P-maximals. $P1$ produces 3 local P-maximals which are BCD : \$90, ABE : \$80, and ACD : \$80. $P2$ produced BE : \$70, and AE : \$60 with support equal to 4 and 4 respectively. All local P-maximals are sent to $P1$ in which any local P-maximal that has any other superset of local P-maximals from other processors are removed. The remaining patterns are declared as global P-maximals (Figure 6.8.E). Subsets of the Global P-maximals that satisfy $Q()$ which is prices > \$30 are kept and others are pruned. The final result set produces D : \$40, E : \$50, AC : \$40, AD : \$50, BC : \$50, BD : \$60, CD : \$70, BE : \$70, AE : \$60, BCD : \$90, ABE : \$80, and ACD : \$80.

6.5 Performance Evaluations

To evaluate our parallel *BifoldLeap* approach, we conducted a set of different experiments to test the effect of pushing *monotone* $Q()$ and *anti-monotone* $P()$ constraints separately, and then both in combination for the same datasets. These experiments were conducted using the same cluster and databases described in 6.3.

We conducted a battery of tests to evaluate the processing load distribution strategy, the scalability vis-à-vis the size of the data to mine, and the speed-up gained from adding

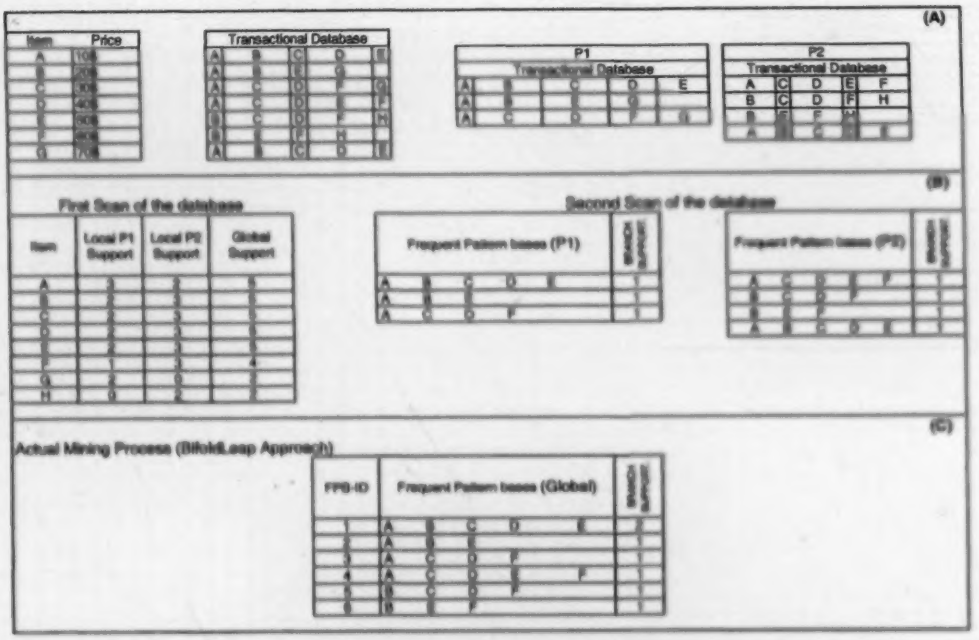


Figure 6.7: Example of parallel *BifoldLeap*: finding the *Frequent-Path-Bases*

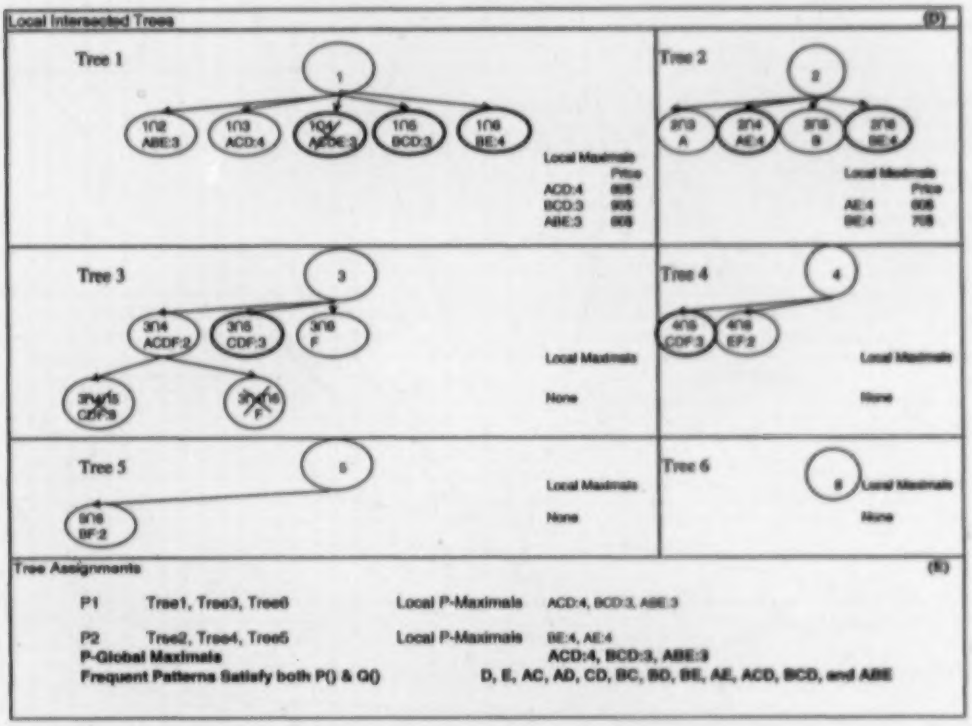


Figure 6.8: Intersecting *Frequent-Path-Bases*

more parallel processing power. Some of the results are portrayed hereafter.

Despite our best efforts and literature searches, we were unable to find any research on parallel constraints-based frequent mining algorithms. For this reason, we could not compare our algorithm against any other algorithms.

For load distribution strategy, we used the same method adopted in Section 6.3.1 as from our experiments in Figure 6.9, we can see that the First-Last distribution gave the best results.

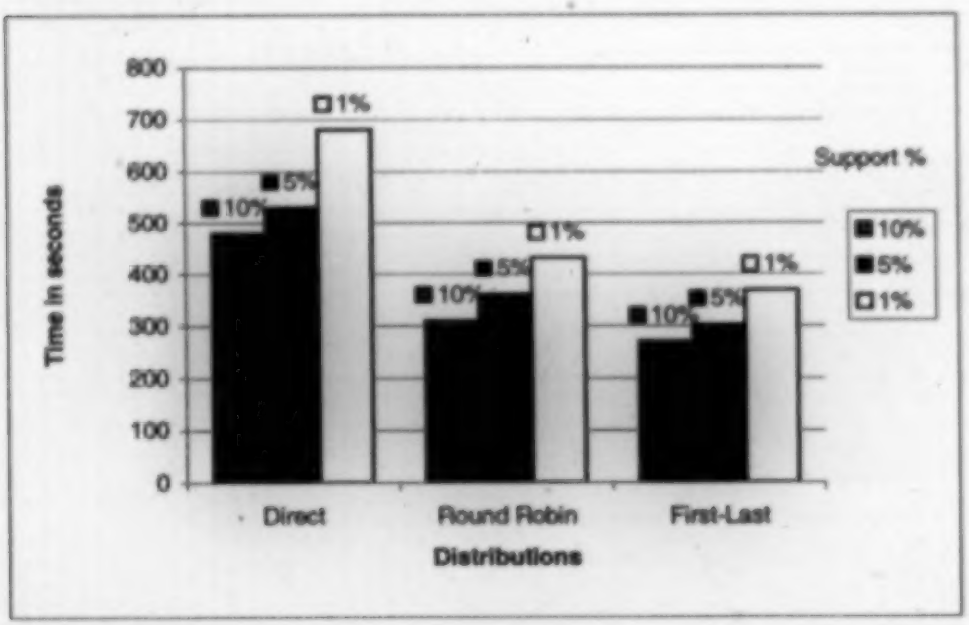


Figure 6.9: Effect of node distributions while pushing both $P()$ and Q constraints. (number of processors = 32)

6.5.1 Scalability with respect to the Database Size

One of the main goals in this work is to mine extremely large datasets. In this set of experiments, we tested the effect of mining different databases made of different transactional databases varying from 100 million transactions up to one billion transactions while pushing both types of constraints $P()$ and $Q()$. Like in the case of absence of constraints in Section 6.3.2, we mined those datasets using 32 processors, with three different support thresholds: 10%, 5% and 1%. We were able to mine one billion transactions in 3700 seconds for a support of 10%, up to 4300 seconds for a support of 1%. Figure 6.10 shows the results of this set of experiments. While the curve does not illustrate a perfect linearity in the scalability, the execution time for the one billion transaction dataset was a very reasonable one hour and eleven minutes with a 1% support and 32 relatively inexpensive processors.

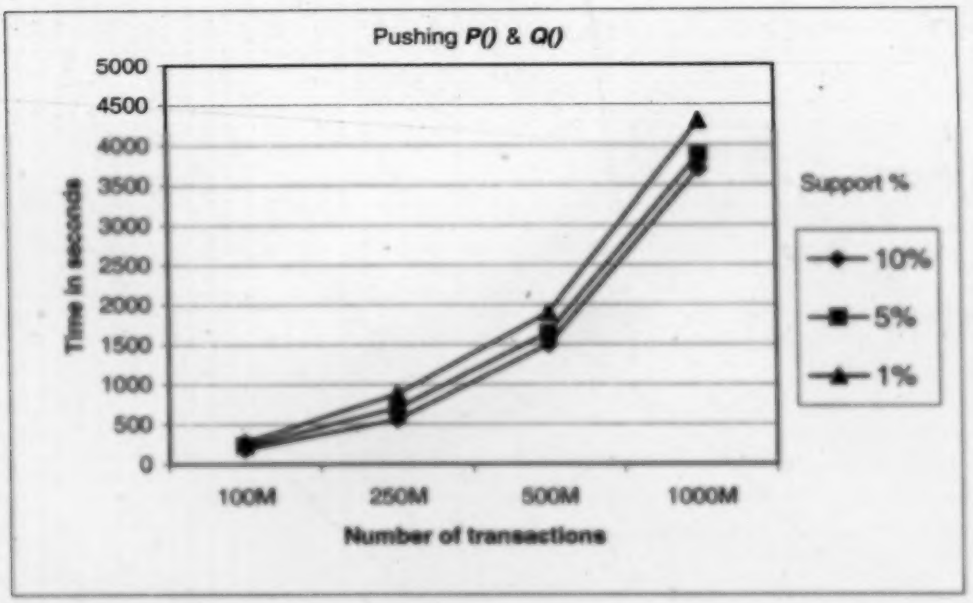


Figure 6.10: Scalability with respect to the transaction size while pushing both $P()$ and Q constraints. (number of processors = 32)

6.5.2 Scalability with respect to the Number of Processors

To test the speed-up of our algorithm with the increase of processors, we fixed the size of the database at 100 million transactions and examined the execution time on this dataset with one to 32 processors. The execution time is reduced sharply when two to four parallel processors are added, and continues to decrease significantly with additional processors (Figure 6.11). Like with Parallel-Leap, the speedup was significant: with 4 processors the speed doubled, with 8 processors it increased four-fold, and with 32 processors we achieved near a 13-fold increase in speed. These results are depicted in Figure 6.12.

6.6 Summary

Parallelizing the search for frequent patterns plays an important role in opening the doors to the mining of extremely large datasets. Not all good sequential algorithms can be effectively parallelized and parallelization alone is not enough. An algorithm has to be well suited for parallelization, and in the case of frequent pattern mining, clever methods for searching are certainly an advantage. The algorithm we propose for parallel mining of frequent patterns while pushing constraints is based on a new technique for astutely jumping within the search space, and more importantly, is composed of autonomous task segments that can be performed separately and thus minimize communication between processors.

Our proposal is based on the finding the *Frequent-Path-Bases*, from which selective jumps

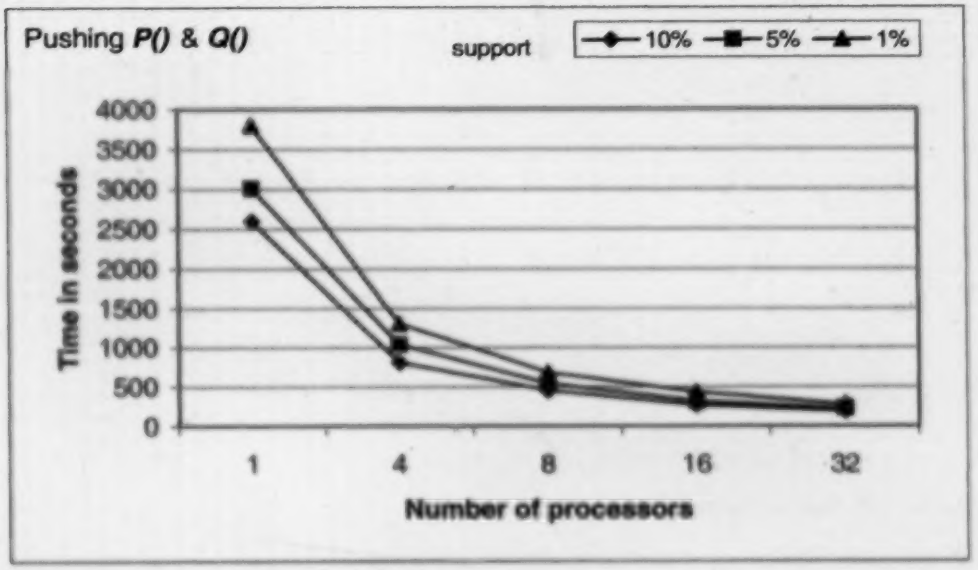


Figure 6.11: Scalability with respect to the number of processors while pushing both $P()$ and Q constraints. (transaction size = 100M)

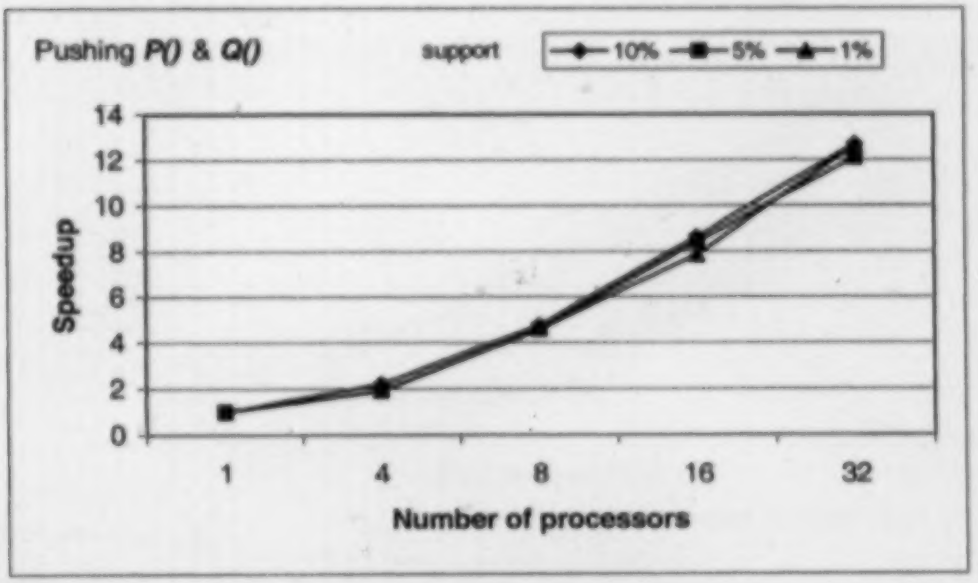


Figure 6.12: Speedup with different support values while pushing both $P()$ and Q constraints. (transaction size = 100M)

in the search space can be performed in parallel and independently from each other *Frequent-Path-Base* in the pursuit of either the maximal patterns or frequent patterns that satisfy user's constraints. The success of this approach is attributed to the fact that *Frequent-Path-Bases* intersection is independent and each intersection tree can be assigned to a given processor. The decrease in the size of intersection trees allows a fair strategy for distributing work among processors and in the course reducing most of the load balancing issues. While other published works claim results with millions of transactions, our approach allows the mining in reasonable time of databases in the order of billion transactions using relatively inexpensive clusters; 16 dual-processor boxes in our case. This is mainly credited to the low communication cost of our approach.

Chapter 7

Conclusions and Future Work

Success isn't how far you got,
but the distance you traveled from where you started.
– Greek Proverb

The end is the crown of any work.
– Russian Proverb

Finding scalable algorithms for frequent itemset mining in extremely large databases is the main goal of our research. To reach this goal, we propose a new set of algorithms that uses novel traversal strategies, and new disk based data structures to search for the set of patterns either sequentially or in parallel and finally to search for subsets of the patterns with the capability of searching for the remaining patterns efficiently without having to revisit the whole database again. Four major issues were addressed to support the central thesis statement of this research.

1. What are the open problems that prevent current algorithms from mining extremely large databases for frequent patterns?

2. Sequential algorithms could help in mining extremely large database using new traversal approaches; approaches that try to avoid some of the redundant work of the existing algorithms.

3. Parallel implementations play an important role in reaching extremely scalable algorithms. Ideally, we need an algorithm specifically designed for parallel execution rather than parallelizing an algorithm designed for sequential execution.

4. The mining process is an interactive process, where the same database is mined more than once for different support thresholds. When mining extremely large databases this becomes an issue, which is why we investigated an interactive approach using a specific transactional database layout.

7.1 Contributions

The main contributions accomplished in this research are parts of a novel framework for mining extremely large databases.

1. COFI-trees: Using these small independent trees we could efficiently mine extremely large database sequentially. We also used these trees to generate the *Frequent-Path-Bases*, which are a presentation of the databases that can be used to generate any set of patterns (all, closed and maximals).

2. Inverted Matrix: A novel data structure that supports interactive mining, where a full scan is not required in many cases while mining the same database. This data structure proved its effectiveness when mining extremely large database, as we need to build it once and mine part of it many times based on the support threshold.

3. Leap-Traversal: Leap-Traversal is a new traversal approach that jumps in the search space among only promising nodes and avoids nodes that would not participate in the answer set and reduces drastically the number of candidate patterns. This approach finds efficiently the set of all, closed and maximal patterns. It also depicts efficient performance in pushing constraints in duality.

4. Parallel-Leap: Mining extremely large databases is still an open problem due to the significance communication cost and number of generated patterns. Parallel-Leap is proposed in this research, allowing the mining of transactional databases in the order of billions of transactions. Different types of frequent patterns can be generated in parallel. Constraints pushing is also shown to be efficiently implemented.

7.2 Future Research

Several directions can be exploited as a continuation of this research. We discuss a few technical challenges in this section and categorize them into two major groups: *Challenges left to explore* and *Future research trends*. In the former, we describe some issues and open questions left to explore in the context of our research mainly challenges for both Inverted Matrix and the Leap approach. In the latter, we point to some technical challenges as evidenced in future research trends for mining for frequent patterns.

7.2.1 Challenges Left to Explore

We describe some challenges and open questions left to explore in the domain of our Leap approach, as follows:

T#	Items				
T1	A	B	C	D	E
T2	A	E	C	H	G
T3	B	C	D	A	E
T4	F	A	H	G	J
T5	A	B	C	E	I
T6	K	A	E	I	C
T7	A	H	E	G	I
T8	K	L	M	N	O
T9	L	R	Q	A	O
T10	P	N	B	A	M

Figure 7.1: Transactional database.

- Inverted Matrix challenges:

  1. **Compressing the size of the Inverted Matrix:**

     Compressing the size of the Inverted Matrix without losing any data is an important issue that could improve the efficiency of the Inverted Matrix algorithm. To achieve this, one could merge similar transactions into new dummy ones, or even merge sub-transactions. For example in Figure 7.1, we can find that the first two transactions contain items A, B, C, D, E and A, E, C, H, G. Ordering both transactions as usual into ascending order according to their frequency produces two new transactions, which are D, B, C, E, A and G, H, C, E, A. Both transactions share the same suffix, which is C, E, and A. Consequently, we can view them as one transaction consisting of $\frac{D,B}{G,H}(C(2), E(2), A(2))$, where any number between parenthesis represents the occurrences of the item preceding it. Using the same methodology we can find that the Inverted Matrix can be compressed by adding an additional field presenting the number of occurrences for the items in the shared suffix. The compressed Inverted Matrix corresponding to Figure 7.1 is depicted in Table 7.1. With this compressed matrix we can dramatically reduce the number of I/Os and thus further improve the performance.

  2. **Reducing the number of I/Os needed**: The Inverted Matrix groups the transactions based on their frequency. Frequent items are clustered at the bottom of the Inverted Matrix. Traversing one transaction can be done by calling more than one page from the database. Reducing the number of pages read

Table 7.1: Compressed Inverted Matrix

loc	Index	Transactional Array		
		1	2	3
1	(P,1)	(10,2)(1)		
2	(F,1)	(5,1)(1)		
3	(Q,1)	(4,1)(1)		
4	(R,1)	(6,2)(1)		
5	(J,1)	(13,2)(1)		
6	(O,2)	(8,2)(1)	(9,2)(1)	
7	(D,2)	(15,1)(2)		
8	(K,2)	(12,2)(1)	(9,1)(1)	
9	(L,2)	(10,1)(1)	(18,1)(1)	
10	(M,2)	(11,1)(1)	(11,2)(1)	
11	(N,2)	(ϕ,ϕ)(1)	(15,2)(1)	
12	(I,3)	(15,1)(1)	(16,1)(1)	(13,3)(1)
13	(G,3)	(14,1)(1)	(14,2)(2)	
14	(H,3)	(16,1)(1)	(17,1)(2)	
15	(B,4)	(16,1)(3)	(18,1)(1)	
16	(C,5)	(17,1)(5)		
17	(E,6)	(18,1)(6)		
18	(A,9)	(ϕ,ϕ)(9)		

from the database by clustering the same transactions on the same pages at the database level is a challenge left to explore.

3. **Updating the Inverted Matrix**: With a horizontal layout, adding transactions is simply appending those transactions to the database. With a vertical layout, each added transaction results in updates in the database entries of all items in the transaction. The Inverted Matrix is neither horizontal nor vertical but a combination, making the addition of new transactions a complex operation. Updating the Inverted Matrix is an important issue in our research. One of the main advantages of the Inverted Matrix is that changing the support level does not mean re-scanning the database again. Changing the database either by adding or deleting new transactions changes the Inverted Matrix, leading to the need of re-building it again. Investigating efficient ways to update the Inverted Matrix without having to rebuild it completely or jeopardizing its integrity is left as a challenge for future research work.

4. **Parallelizing the construction of Inverted Matrix**: The Inverted Matrix could be built in parallel. Each processor could build its own Inverted Matrix that reflects all transactions on its node in the cluster. The index part of the small Inverted Matrices would reflect the global frequency of the items in all transactions. Building these distributed Inverted Matrices could also be done using two passes over the local data. The first pass or scan could generate the

local frequency for each item. Generating the global frequency of each item could be done either by broadcasting or scattering these local supports. The second pass for each local node is almost identical to the second pass of the sequential version, where communication between nodes is minimal.

- Leap approach challenges:

  1. Investigate the possibility of applying more pruning ideas.

  2. Intersection of trees: currently we are using a bitmap approach where intersecting two nodes is applying the AND operation. Other approaches could also be investigated.

7.2.2 Future Research Trends

Here we discuss some future research trends in the context of frequent pattern mining. These points are as follows:

- What algorithm is considered the best? This is still an open question. While many researchers are still investigating this issue and workshops are dedicated to answer this question, clear winners for all cases still cannot be found. However, most of these workshops ignored the fact that the behaviour of most existing algorithms changes completely once they start to mine extremely large databases.

- How to benchmark technical solutions? Benchmarking is as important as the solutions themselves. Yet many fields still lack any type of rigorous evaluation. Performance benchmarking has always been an important issue in databases and has played a significant role in the development, deployment and adoption of technologies [76]. To help assess the myriad of algorithms for frequent itemset mining, we built an open framework and testbed to analytically study the performance of different algorithms and their implementations [36], and contrast their achievements given different data characteristics, different conditions, and different types of patterns to discover their constraints. This facilitated reporting consistent and reproducible performance results using known conditions. Further work is definitely necessary, because we barely scratched the surface and we still have no means to identify a winner algorithm under different conditions.

- How to determine database characteristics? Given different characteristics of the database, a different algorithm could be pronounced as a winner with regard to space and time complexity. The true characteristics of the database that could determine this winner are still unknown. In the context of [36] we identified some characteristics

such as the average transaction length, the size of transactions, support used, and the ordered list of frequent 1-itemset. We call this as the database signature, as well as the power low distribution of the transactions size. However, to build an accurate supervised classifier to determine the most appropriate algorithm given a dataset, we found that the size of maximals are the most discriminative. This defeats the purpose since to get the maximals, one needs to run the algorithms. We propose to sample a database, extract the maximals form the sample using any algorithm such as FP-MAX or COFI-MAX and generate samples to determine the appropriate algorithm for the database on hand. This idea could be exploited in a parallel environment in which each processor, after receiving its partition to mine, could independently determine the best algorithm to use on its own for mining frequent patterns.

- What patterns to generate? We may need to change the way we look at the problem, as we may not need to generate the whole answer set at once. When we search for patterns (all, closed, and maximals) in general millions of patterns are generated. Understanding them requires another mining tool. It is important to investigate other types of patterns, such as the usefulness of having *Frequent-Path-Bases* and maximal sets only, where any needed pattern to test can be generated on the fly as a subset of an existing maximal. Its support can be generated from the *Frequent-Path-Bases*. The data mining community started lately to investigate this issue, which is finding a summary of the patterns where the remaining can be generated later, once needed. The student paper runner up award paper [81] for the SIGKDD 2005 conference [5] investigated the issue of using patterns similar to the *Frequent-Path-Bases* to find the support of any patterns if needed.

7.3 Final Thoughts

We are captivated by the idea of finding knowledge in data. Huge databases with billions of transactions contain important patterns if we can find them, and these patterns lead to relevant, applicable information and knowledge. Databases with billions of records or transactions are not a rare occurrence nowadays. Frequent pattern mining opens the door to a greater understanding of human activity and all those things we track with data.

We have not invented a wheel here. We only hope that we have made the wheel smoother so it can do its work a little faster. As Sir Isaac Newton said, "*If I have seen farther than others, it is because I was standing on the shoulders of giants*". Our goal with this research has been to take the good work done by our academic predecessors and take it to the next level. We gratefully acknowledge the work of researchers on frequent pattern mining. These are the shoulders upon which we stand.

Bibliography

[1] R. Agrawal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *In 7th Int'l Conference on Knowledge Discovery and Data Mining*, 2000.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[4] Maria-Luiza Antonie and Osmar R. Zaïane. Text document categorization by term association. In *IEEE International Conference on Data Mining*, pages 19–26, December 2002.

[5] KDD 2005 Awards. Proc. 2005 int'l conf. on data mining and knowledge discovery (acm sigkdd). http://www.acm.org/sigs/sigkdd/kdd2005/awards.html.

[6] Jay Ayres, J. E. Gehrke, Tomi Yiu, and Jason Flannick. Sequential pattern mining using bitmaps. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, Alberta, Canada, July 2002.

[7] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern information retrieval. In *ACM Press, NY*, 1999.

[8] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD*, 1998.

[9] F. Beil, M. Ester, and X. Xu. Frequent term-based text clustering. In *Proc. 8th Int. Conf. on Knowledge Discovery and Data Mining (KDD '2002), Edmonton, Alberta, Canada*, 2002.

[10] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Examiner: Optimized level-wise frequent pattern mining with monotone constraints. In *IEEE ICDM*, Melbourne, Florida, November 2004.

[11] F. Bonchi and Bart Goethals. Fp-bonsai: the art of growing and pruning small fp-trees. In *In Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, 2004.

[12] Christian Borgelt. Apriori implementation. http://fuzzy.cs.uni-magdeburg.de/~borgelt/apriori/apriori.html.

[13] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings ACM SIGMOD International Conference on Management of Data*, 1997.

[14] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *In Proc. of the ACM SIGMOD Conference on Management of Data*, 1997.

[15] Cristian Bucila, Johannes Gehrke, Daniel Kifer, and Walker White. Dualminer: A dual-pruning algorithm for itemsets with constraints. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 42–51, Edmonton, Alberta, August 2002.

[16] Douglas Burdick, Manuel Calimlim, and Johannes Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001.

[17] S. Chaudhuri. Data mining and database systems: Where is the intersection? In *Bulletin of the Technical Committee on Data Engineering, 21*, 1998.

[18] Jiyang Chen, Mohammad El-Hajj, Osmar R. Zaïane, and Randy Goebel. Constraint-based mining with visualization of web page connectivity and visit associations. In *Submitted to Proc. of the IEEE 2006 International Conference on Data Mining (ICDM06)*, 2006.

[19] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on a shared-memory multi-processors. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures, ACM Press, NY*, pages 279 – 288, 1998.

[20] David Wai-Lok Cheung, Jiawei Han, Vincent Ng, Ada Wai-Chee Fu, and Yongjian Fu. A fast distributed algorithm for mining association rules. In *PDIS*, pages 31–42, 1996.

[21] Soon M. Chung and Congnan Luo. Parallel mining of maximal frequent itemsets from databases. In *15th IEEE International Conference on Tools with Artificial Intelligence*, 2003.

[22] Chad Creighton, , and Samir Hanash. Mining gene expression databases for association rules. In *Journal of Bioinformatics Vol. 19 no. 1*, 2003.

[23] S. Downs and M. Wallace. Mining association rules from a pediatric primary care decision support system. In *Proc American Medical Informatics Association Annual Symposium*, 2000.

[24] Mohammad El-Hajj and Osmar R. Zaïane. Cofi-tree mining: A new approach to pattern growth with reduced candidacy generation. In *in Workshop on Frequent Itemset Mining Implementations (FIMI'03) in conjunction with IEEE-ICDM*, November 2003.

[25] Mohammad El-Hajj and Osmar R. Zaïane. Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. In *In Proc. 2003 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August 2003.

[26] Mohammad El-Hajj and Osmar R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *In Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, September 2003.

[27] Mohammad El-Hajj and Osmar R. Zaïane. Parallel association rule mining with minimum inter-processor communication. In *Fifth International Workshop on Parallel and Distributed Databases (PaDD'2003) in conjunction with the 14th Int' Conf. on Database and Expert Systems Applications DEXA2003*, September 2003.

[28] Mohammad El-Hajj and Osmar R. Zaïane. Cofi approach for mining frequent itemsets revisited. In *9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD-04)*, June 2004.

[29] Mohammad El-Hajj and Osmar R. Zaïane. Finding all frequent patterns starting from the closure. In *The First International Conference on Advanced Data Mining and Applications (ADMA)*, July 2005.

[30] Mohammad El-Hajj and Osmar R. Zaïane. Implementing leap traversals of the itemset lattice. In *in ACM SIGKDD Open Source Data Mining Workshop on Frequent Pattern Mining Implementations (OSDM'05)*, August 2005.

[31] Mohammad El-Hajj and Osmar R. Zaïane. Mining with constraints by pruning and avoiding ineffectual processing. In *The 18th Australian Joint Conference on Artificial Intelligence*, December 2005.

[32] Mohammad El-Hajj and Osmar R. Zaïane. On frequent itemset mining with closure. In *2nd International Conference on Information Technology (ICIT 2005)*, May 2005.

[33] Mohammad El-Hajj and Osmar R. Zaïane. Yafima: Yet another frequent itemset mining algorithm. In *Journal Of digital information management 3(4)*, December 2005.

[34] Mohammad El-Hajj and Osmar R. Zaïane. Parallel bifold: Large-scale parallel pattern mining with constraints. In *Distributed and Parallel Databases. An International Journal, by Springer.*, 2006.

[35] Mohammad El-Hajj and Osmar R. Zaïane. Parallel leap: Large-scale maximal pattern mining in a distributed environment. In *nternational Conference on Parallel and Distributed Systems (ICPADS'06)*, July 2006.

[36] Mohammad El-Hajj, Osmar R. Zaïane, Yi Li, and Stella Luk. Scrutinizing frequent pattern discovery performance. In *IEEE ICDE*, April 2005.

[37] Mohammad El-Hajj, Osmar R. Zaïane, and Paul Nalos. Bifold constraint-based mining by simultaneous monotone and anti-monotone checking. In *The fifth IEEE International Conference on Data Mining (ICDM'05)*, November 2005.

[38] M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions*. Wiley, 2000.

[39] R. Feldman and H. Hirsh. Mining associations in text in the presence of background knowledge. In *Proc. 2st Int. Conf. Knowledge Discovery and Data Mining*, pages 343–346, Portland, Oregon, Aug. 1996.

[40] A.A. Freitas. A survey of parallel data mining. In *Proc. 2nd Int. Conf. on the Practical Applications of Knowledge Discovery and Data Mining*, pages 287–300, 1998.

[41] D. Gamberger, N. Lavrac, and V. Jovanoski. High confidence association rules for medical diagnosis. In *Intelligent Data Analysis in Medicine and Pharmacology, (IDAMAP'99)*, Washington, DC, November 1999.

[42] B. Goethals. Survey on frequent pattern mining. Manuscript, 2003.

[43] Bart Goethals and Mohammed Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *in Workshop on Frequent Itemset Mining Implementations (FIMI'03) in conjunction with IEEE-ICDM*, 2003.

[44] Karam Gouda and Mohammed Javeed Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[45] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI'03, Workshop on Frequent Itemset Mining Implementations*, November 2003.

[46] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *In ACM SIGMOD Conf. Management of Data*, 1997.

[47] E-H. Han, G. Karypis, and V.Kumar. Scalable parallel data mining for association rule. *Transactions on Knowledge and data engineering*, 12(3):337–352, May-June 2000.

[48] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[49] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman, San Francisco, CA, 2001.

[50] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[51] IBM_Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.

[52] Renta Ivncsy and Istvn Vajk. A survey of discovering frequent patterns in graph data. In *Proc. of the IASTED International Conference on Databases and Applications DBA 2005 as part of the Twenty-Third IASTED International Multi-Conference on Applied Informatics*, pages 53–58, Innsbruck, Austria, Feb 2005. Acta Press.

[53] George K. *Zipf, Human Behaviour and the Principle of Least-Effort*. Addison-Wesley, Cambridge MA, 1949.

[54] M. Klemettinen, H. Mannila, and H. Toivonen. Rule discovery in telecommunication alarm data. In *Journal of Network and Systems Management, 7(4)*, 1999.

[55] L.V. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *In proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'99)*, pages 157–168, 1999.

[56] Wenmin Li, Jiawei Han, and Jian Pei. Cmar: accurate and efficient classification based on multipleclass-association rules. In *First IEEE International Conference on Data Mining ICDM*, 2001.

[57] W. Lin, S. A. Alvarez, , and C. Ruiz. Efficient adaptive-support association rule mining for recommender systems. In *Data Mining and Knowledge Discovery*, pages 6(1):83 – 105, January 2002.

[58] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *4th Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York City, NY, August 1998.

[59] Junqiang Liu, Yunhe Pan, Ke Wang, and Jiawei Han. Mining frequent item sets by oppotunistic projection. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 229–238, Edmonton, Alberta, August 2002.

[60] Heikki Mannila. Inductive databases and condensed representations for data mining. In *International Logic Programming Symposium*, 1997.

[61] R. Ng, L.V. Lakshmanan, J. Han, and T. Mah. Exploratory mining via constraint frequent set queries. In *In A Delis, SIGMOD 99*, pages 556–558, 1999.

[62] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[63] S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *In ACM Intl. Conf. Infomration and Knowledge Management*, 1995.

[64] Srinivasan Parthasarathy, Mohammed Zaki, Mitsunori Ogihara, and Wei Li. Parallel data mining for association rules on shared-memory systems. In *in Knowledge and Information Systems, Volume 3, Number 1*, 2001.

[65] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *International Conference on Database Theory (ICDT)*, pages pp 398–416, January 1999.

[66] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448, 2001.

[67] J. Pie and J. Han. Can we push more constraints into frequent pattern mining? In *In ACM SIGKDD Conference*, pages 350–354, 2000.

[68] J. Pie, J. Han, and L.V. Lakshmanan. Mining frequent itemsets with convertible constraints. In *In ICDE 2001 Conference*, pages 433–442, 2001.

[69] Iko Pramudiono and Masaru Kitsuregawa. Tree structure based parallel frequent pattern mining on pc cluster. In *DEXA*, pages 537–547, 2003.

[70] Agrawal R. and Shafer J. Parallel mining of association rules. In *IEEE Transactions in Knowledge and Data Eng.*, pages 962–969, 1996.

[71] Srikant R. and Agrawal R. Mining generalized association rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, 1995.

[72] A. Rungsawang, A. Tangpong, P. Laohawee, and T. Khampachua. Novel query expansion technique using apriori algorithm. In *TREC, Gaithersburg, Maryland*, 1999.

[73] Ashok Savasere, Edward Omiecinski, , and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *In Proceedings of the 21st International Conference on Very Large Data Bases*, pages 432–444, 1995.

[74] Takahiko Shintani and Masaru Kitsuregawa. Hash based parallel algorithms for mining association rules. In *Proceedings of IEEE Fourth International Conference on Parallel and Distributed Information Systems*, 1996.

[75] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns form web data. In *SIGKDD Explorations, 1(2)*, January 2000.

[76] Yannis Theodoridis. Ten benchmark database queries for location-based services. *The Computer Journal*, 46(6):713–725, 2003.

[77] Roger M.H. Ting, James Bailey, and Kotagiri Ramamohanarao. Para dualminer: An efficient parallel implementation of the dualminer algorithm. In *Eight Pacific-Asia Conference, PAKDD 2004*, pages 96–105, Sydney, Australia, May 2004.

[78] Jianyong Wang, Jiawei Han, and Jian Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'03)*, Washington, DC, USA, 2003.

[79] John Wang. *Data Mining: Opportunities and Challenges*. Idea Group Inc (IGI), 2003.

[80] S. Ben Yahia, T. Hamrouni, and E. M. Nguifo. Frequent closed itemset based algorithms: A thorough structural and analytical survey. *ACM SIGKDD Explorations*, 8(1):91–104, June 2006.

[81] Xifeng Yan, Hong Cheng, Dong Xin, and Jiawei Han. Summarizing itemset patterns: A profile-based approach. In *in Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August 2005.

[82] Osmar R. Zaïane and Mohammad El-Hajj. Advances and issues in frequent pattern mining (conference tutorial notes). In *Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, May 2004.

[83] Osmar R. Zaïane and Mohammad El-Hajj. Pattern lattice traversal by selective jumps. In *in Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August 2005.

[84] Osmar R. Zaïane and Mohammad El-Hajj. Bi-directional constraint pushing in frequent pattern mining. In *Data Mining Patterns: New Methods and Applications, A book to be published by Idea Group Inc*, 2007.

[85] Osmar R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast parallel association rule mining without candidacy generation. In *in Proc. of the IEEE 2001 International Conference on Data Mining (ICDM01)*, San Jos, CA, USA, December 2001.

[86] Osmar R. Zaïane, Jiawei Han, and Hua Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

[87] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, 1997.

[88] Mohammed Zaki and C-J. Hsiao. ChARM: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data Mining*, April 2002.

[89] Mohammed J. Zaki and Karam Gouda. Fast vertical mining using diffsets. Technical Report Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute, 2001.

[90] Zijian Zheng, Ron Kohavi, and Llew Mason. Real world performance of association rule algorithms. In *7th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 401–406, 2001.